An Oracle White Paper
August 2010

Beginner's Guide to Oracle Grid Engine 6.2

ORACLE®

Executive Overview

This white paper will walk through basic installation and configuration of Oracle Grid Engine 6.2, explain Oracle Grid Engine architectural concepts, and present some suggested configurations for clusters of various sizes. The target audience is system administrators who have downloaded the Oracle Grid Engine 6.2 evaluation version or the Grid Engine 6.2 courtesy binaries or who have just purchased the Oracle Grid Engine 6.2 software.

Introduction

The Oracle Grid Engine software is a distributed resource management (DRM) system that enables higher utilization, better workload throughput, and higher end-user productivity from existing compute resources. By transparently selecting the resources that are best suited for each segment of work, the Oracle Grid Engine software is able to distribute the workload efficiently across the resource pool while shielding end users from the inner working of the compute cluster.

In order to facilitate the most effective and efficient scheduling of workload to available resources, the Oracle Grid Engine software gives administrators the ability to accurately model as a resource any aspect of the compute environment that can be measured, calculated, specified, or derived. The Oracle Grid Engine software can monitor and manage resources that are concrete, such as CPU cores or system memory, as well as abstract resources, like application licenses or mounted file systems.

In addition to allocating resources based on workload requirements, the Oracle Grid Engine software provides an arsenal of advanced scheduling features that allow an administrator to model not only the compute environment, but also the business rules that control how the resources in that compute environment should be allocated. Some of the advanced scheduling features offered by the Oracle Grid Engine 6.2 software are advance reservation of resources, fair-share scheduling, scheduling by resource value, observation of deadline times

for specific segments of work, user-specified relative priorities, and starvation prevention for resource-hungry workload segments.

Because of the flexibility offered by the Oracle Grid Engine software, administrators sometimes find themselves in need of some help getting started. This white paper endeavors to fill that need by providing guidance on the following topics:

- Chapter 1, "Introduction to Oracle Grid Engine," describes some basic Oracle Grid Engine concepts.

- Chapter 2, "Oracle Grid Engine Scheduler," introduces the complete list of scheduling policies and what problems each one solves.

- Chapter 3, "Planning an Oracle Grid Engine Installation," provides suggestions for preparing the computing environment for a cluster installation.

- Chapter 4, "Suggested Oracle Grid Engine Configurations," presents some common cluster types with suggested configuration options.

This white paper is intended for administrators who will be or are in the process of installing an Oracle Grid Engine cluster and for anyone who wishes to understand more about the deployment and configuration of Oracle Grid Engine clusters. It is meant as a companion and supplement to the Oracle Grid Engine 6.2 Installation Guide[1].

---

[1]Oracle Grid Engine Installation Guide: http://wikis.sun.com/display/gridengine62u6/Installing

Chapter 1: Introduction to Oracle Grid Engine

The Oracle Grid Engine software is a distributed resource management (DRM) system designed to maximize resource utilization by matching incoming workload to available resources according to the needs of the workload and the business policies that are in place. This chapter provides an overview of some basic Oracle Grid Engine concepts, the Oracle Grid Engine component architecture, and the scheduling policies.

Oracle Grid Engine Jobs

Each incoming segment of work is known as a *job*. Administrators and end users submit jobs using `qmon`, the graphical user interface; one of the command-line tools, such as `qsub` or `qrsh`; or programmatically by using the Distributed Resource Management Application API[2] (DRMAA, pronounced like "drama").

Each job includes a description of what to do, for example an executable command, and a set of property definitions that describe how the job should be run. An example job might be to run the script, `myjob.sh`, with the output stream redirected to `/var/tmp/run1` and the working directory set to `/home/dant`. In this example, the Oracle Grid Engine software would locate an appropriate free resource on which to run the `myjob.sh` script and send the job there to be executed.

The Oracle Grid Engine software recognizes four basic classes of jobs: batch jobs, parametric (or array) jobs, parallel jobs, and interactive jobs.

• A traditional *batch* job is single segment of work that is executed only once, as in the above `myjob.sh` example.

• A *parametric job*, also known as an *array job*, consists of a series of workload segments that can all be run in parallel but are completely *independent* of one another. All of the workload segments of an array job, known as *tasks*, are identical except for the data sets on which they operate. Distributed rendering of visual effects is a classic example of an array job.

• A *parallel* job consists of a series of cooperating tasks that must all be executed at the same time, often with requirements about how the tasks are distributed across the resources. Very often parallel jobs make use of a parallel environment, such as MPI, to enable the tasks to intercommunicate.

• An *interactive* job provides the submitting user with an interactive login to an available resource in the compute cluster. Interactive jobs allow users to execute work on the compute cluster that is not easily submitted as a batch job.

Oracle Grid Engine Component Architecture

The following diagram shows the components of an Oracle Grid Engine cluster at a high level:

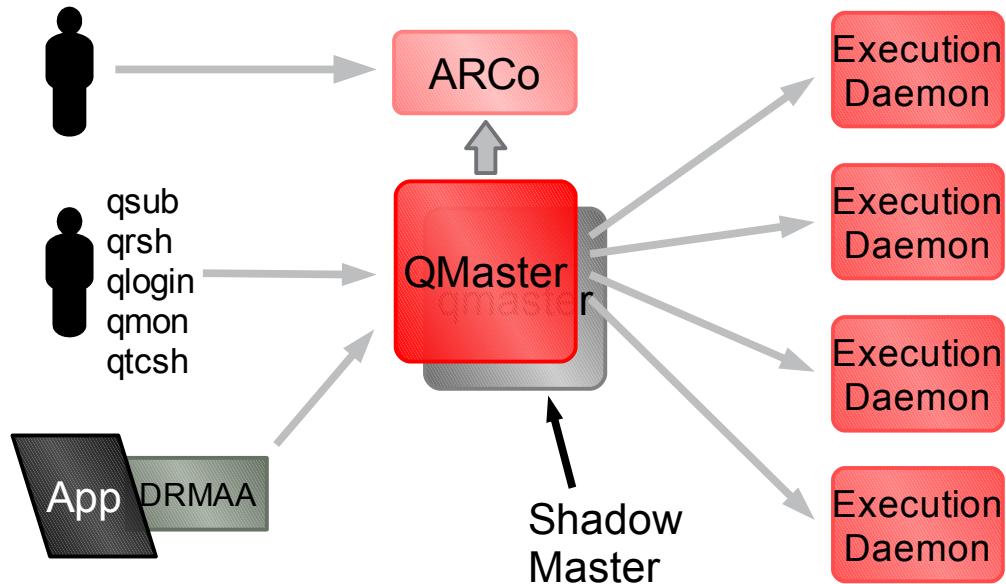[2]DRMAA Working Group Home Page: http://www.drmaa.org/

Figure 1: Oracle Grid Engine Component Architecture

At the center of the above diagram is the *qmaster*. The qmaster is the central component of an Oracle Grid Engine compute cluster, accepting incoming jobs from users, assigning jobs to resources, monitoring the overall cluster status, and processing administrative commands. The qmaster is a multi-threaded daemon that runs on a single host in the compute cluster. To reduce unplanned cluster downtime, one or more *shadow masters* may be running on additional nodes in the cluster. In the case that the qmaster or the host on which it is running fails, one of the shadow masters will promote the host on which it is running to the new qmaster node by locally starting a new qmaster daemon.

Each host in the cluster that is to execute jobs will host an *execution daemon*. The execution daemon receives jobs from the qmaster and executes them locally on its *host*. The capacity to execute a job is known as a job *slot*. The Oracle Grid Engine software does not place a limit on the number of job slots that an execution daemon can offer, but in most cases the number of job slots is determined by the number of CPU cores available at the execution host. When a job has completed, the execution daemon notifies the qmaster so that a new job can be scheduled to the now free job slot. At a fixed interval each execution daemon will send a report of its status to the qmaster. If the qmaster fails to receive several consecutive load reports from an execution daemon, the qmaster will mark that execution host and all its resources as no longer available and will remove it from the list of available job scheduling targets.

Jobs are sent to the qmaster in a variety of ways. DRMAA provides a programmatic interface for applications to submit, monitor, and control jobs. The Oracle Grid Engine software comes with C and Java™ language DRMAA bindings, making it possible to use DRMAA from a wide range of applications. `qmon` is the Oracle Grid Engine graphical user interface. From `qmon` users and administrators can submit, monitor, and control jobs as well as manage all aspects of the cluster. `qsub` is a command-line utility for submitting batch, array, and parallel jobs. `qsh`, `qrsh`, `qlogin`,

and `qtcsh` are all command-line utilities for submitting various kinds of interactive jobs. For more information on these command-line utilities, see the Oracle Grid Engine 6.2 Administration Guide[3].

The last component shown in the diagram is ARCo, the Accounting and Reporting Console. ARCo is a web-based tool for accessing Oracle Grid Engine accounting information automatically stored in an SQL database. Using ARCo, end users and administrators can create and run queries against the cluster's accounting data. Administrators can also pre-configure ARCo with common queries for their end users to run.

Oracle Grid Engine Basics

To give administrators the flexibility to model even the most complex compute environments, the Oracle Grid Engine software provides powerful abstractions for resource and cluster modeling. Three of those abstractions, *queues*, *host groups*, and *resources*, are discussed below.

Queues

In an Oracle Grid Engine cluster, queues define where and how jobs are executed. An Oracle Grid Engine queue is a logical abstraction that aggregates a set of job slots across one or more execution hosts. A queue also defines attributes related to how jobs are executed in those job slots, such as the signal used to suspend a job or the executable that is run before a job to prepare the job's execution environment. Finally, a queue defines attributes related to policy, such as which users are allowed to run jobs in that queue or when the queue has become overloaded.

For example, a default Oracle Grid Engine installation will have a queue called `all.q`. `all.q` will span every host in the cluster, with a number of job slots on each host equal to the number of CPU cores available at that host. By default, `all.q` will allow jobs from any user. If a user submits a job to such a default cluster, the job will run in one of `all.q`'s job slots on the execution host that is least loaded.

An Oracle Grid Engine queue is composed of a series of queue instances, one per execution host. Queue instances are named for the queue and execution host with which they are associated, such as `all.q@myhost`. Each queue instance inherits all the properties of the queue to which it belongs. Each queue instance can override inherited properties with values that are specific to its execution host. For example, in a default cluster as described above, the queue, `all.q`, will have a slots attribute that is set to 1. This means that `all.q` offers one job slot on every host in the queue. Each queue instance in `all.q`, however, will override that slots value with a value equal to the number of CPU cores on that execution host. In this way, `all.q` is able to offer a number of job slots on each execution host that is appropriate for that execution host.

---

[3]Oracle Grid Engine Administration Guide: http://wikis.sun.com/display/gridengine62u6/Administering

Cluster Queues

In the 5.3 and prior releases of the Oracle Grid Engine software, a queue was limited to a single host. The 6.0 release introduced cluster queues, which were a new kind of queue that was able to span multiple hosts. The 5.3 definition of a queue was roughly equivalent to the 6.0 definition of a queue instance.

At the transition point from 5.3 to 6.0, the term "queue" referred to a 5.3 queue, and the term "cluster queue" referred to a 6.0 multi-host queue. Today, many years later, the term "queue" refers to the 6.0 cluster queue, and the term "cluster queue" has passed out of the vernacular. Queue instances are known today as "queue instances."

Be aware, however, that in some older documentation and papers, circa 2004 or earlier, the queues named in the documentation may not be the queues used by the Oracle Grid Engine software today.

Host Groups

An Oracle Grid Engine host group is an administrative handle for referencing multiple execution hosts at once. Host groups have no attributes associated with them other than the list of hosts they contain. Host groups can be hierarchical, with each host group potentially containing a mix of hosts and other host groups. For example, a default Oracle Grid Engine installation will have a host group called `@allhosts` that contains every execution host in the cluster. Instead of listing all the execution hosts in the cluster explicitly (and redundantly) `all.q` is defined as running across `@allhosts`.

The following figure shows a graphical representation of what a sample queue and host group configuration might be:
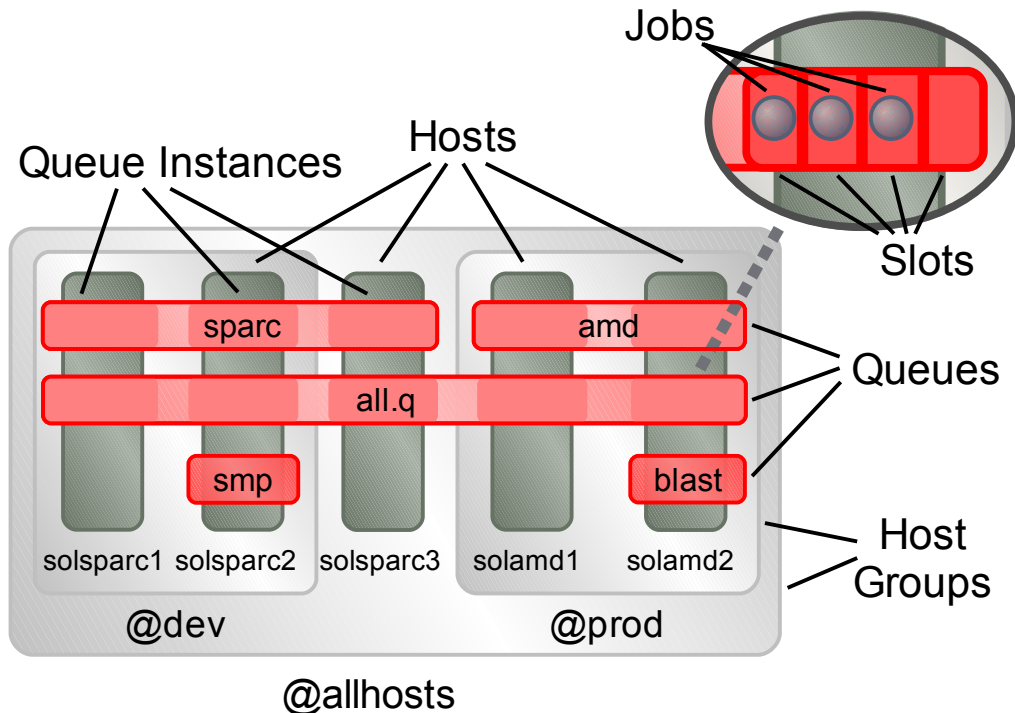

Figure 2: Example Host Group and Queue Configuration

[4]NCBI BLAST Home Page: http://www.ncbi.nlm.nih.gov/BLAST/

Resources

An Oracle Grid Engine resource, also known as a *complex*, is an abstraction that models some property of an execution host or the compute environment. Examples include the load average of an execution host, the number of application licenses for a particular application that are available for use by jobs, or the name of the data center that houses an execution host. When a user submits a job, he or she can include resource needs in that job's definition. Resources are then used by the scheduler to match jobs to execution hosts and by the administrator to create usage policies. Scheduling is explained in greater detail in Chapter 2. Using resource definitions, administrators can model anything in their compute environment that can be defined, tracked, and/or measured. Resources can even be used to model abstract notions, such as importance or job type. Oracle Grid Engine recognizes three basic types of resources: *static*, *consumable*, and *predefined*.

Static resources model a fixed characteristic of an execution host or the compute environment. They have numeric or text values that are assigned by the administrator. For example, the version of the Java Virtual Machine installed on an execution host might be modeled as a static resource with a numeric value.

There is a special type of static resource called a *dynamic* resource. The Oracle Grid Engine software allows an administrator to create executable programs or scripts (known as *load sensors*) that measure the values of resources. An execution daemon periodically runs those programs and reports the measured resource values.

Consumable resources (also known as *consumables*) model elements of the compute environment that are available in fixed quantity and that are "consumed" by running jobs. When a job requests a consumable resource, the amount of that resource that is available is decremented for the duration of that job. When a job ends, all of the consumable resources it was using are freed for use by other jobs. When a consumable resource is exhausted, no more jobs requesting that resource can be scheduled until more of that resource becomes available again. Application software licenses are classic consumable resources. It is important that no more jobs are started than there are appropriate licenses available.

The *predefined* resources are a set of resources automatically measured and reported by every execution daemon. These include things like the system load and the amount of free memory.

Resources are associated with either a queue, a host, or all hosts (called the *global host*). A resource that is assigned to a queue is available from every queue instance in that queue. A resource that is assigned to a host is available from all queue instances on that host. A resource that is assigned to the global host is available from all hosts and all queues.

For consumable resources, the resource consumption is counted at the level where the resource is assigned. For example, if a consumable is assigned at the host level, that resource's consumption is counted for all queue instances on the host all together. When the resource is consumed from one queue instance, it reduces the amount of that resource available for all the queue instances on that host. If a resource is defined at more than one level, it is tracked at all levels where it is defined. At any level, the most restrictive value applies. For example, if a consumable is assigned at the global host level with a value of 10, at host A with a value of 20, and at host B with a value of 5, there will be 10 of that

consumable available to be shared among all hosts in the cluster. Only 5 of that consumable, however, can be in use at a time on host B. Even though host A defined the consumable with a higher value than at the global host, that host is still restricted to the global value of 10. A host cannot offer more of a resource than is available globally.

Other Concepts

The Oracle Grid Engine software uses several other abstractions for modeling other parts of a cluster. In brief, they are:

- **Projects** – used for access control, accounting, and the ticket policies discussed in the next chapter. Projects provide a way to aggregate together jobs that serve a common purpose.

- **Users** – used for the ticket policies discussed in the next chapter.

- **Users lists** – used for managing access control and the ticket policies discussed in the next chapter, user lists come in two flavors, *access control lists* and *departments*. Access control lists (ACLs) are used to grant or deny access to resources and privileges. Departments are only used for the ticket policies and accounting and provide a way to aggregate together users in a common organization.

- **Parallel Environments** – used by parallel jobs to start and stop the parallel framework used by a job to synchronize and communicate among slave tasks. MPI is a common example.

- **Checkpointing Environments** – used by jobs of any type to enable checkpointing through an external checkpointing facility.

- **Execution Hosts** – represent the configuration of an execution host's stateful characteristics.

- **Host Configurations** – represent the configuration of an execution host's behavioral characteristics.

- **Calendars** – used to enable, disable, suspend, and resume queues on a predefined schedule.

Chapter 2: Oracle Grid Engine Scheduler

At the heart of an Oracle Grid Engine cluster is the scheduler. It is responsible for prioritizing pending jobs and deciding which jobs to schedule to which resources.

In previous versions of the Oracle Grid Engine software, the scheduler was a separate daemon that communicated directly with the qmaster. Starting with the Oracle Grid Engine 6.2 release, the scheduler is simply a part of the qmaster. Some documentation (including this paper) may still refer to "the scheduler" as though it is an individual entity, but what is really meant is "the scheduling function of the qmaster."

When a user submits a job to an Oracle Grid Engine cluster, the job is placed in a pending list by the qmaster. Periodically a scheduling run will be triggered, which will attempt to assign the highest priority jobs in the pending list to the most appropriate available resources. A scheduling run can be triggered in three ways. First, scheduling runs are triggered at a fixed interval. In a default Oracle Grid Engine installation, scheduling runs are triggered every fifteen seconds. Second, a scheduling run may be triggered by new job submissions or by notification from an execution daemon that one or more jobs have finished executing. In a default Oracle Grid Engine installation, this feature is not enabled. Third, an administrator can trigger a scheduling run explicitly through the `qconf -tsm` command, which can be useful when troubleshooting the cluster.

Scheduler Settings

When installing the Oracle Grid Engine qmaster, the choice is offered among "Normal," "High," and "Max" scheduling settings. The normal and high settings cause the scheduler to run at a fixed fifteen-second interval with immediate scheduling disabled. The max setting causes the scheduler to run at a two-minute fixed interval with additional scheduler runs four seconds after any job is submitted of ends.

The process of scheduling a job has two distinct stages, which are described in detail below. In the first stage, job selection, the qmaster sorts the jobs in order of priority. Job priority is derived from scheduler policies. A job may receive priority because of who submitted it, what resources the job needs, by when the job must be started, how long the job has been waiting to start, or for a variety of other reasons. In the second stage, job scheduling, the sorted list of pending jobs is assigned to job slots in order of priority. If there are more jobs in the pending list than there are free job slots, the lower priority jobs will remain pending until the next scheduling run.

Job Selection

Job selection is the first stage of the scheduling process, in which every job in the pending job list is assigned a priority, and the entire list of sorted according to priority order. The assignment of priority is driven by the active set of scheduling policies. The Oracle Grid Engine qmaster provides a series of scheduling policies to allow administrators to model the business rules that govern the use of the cluster's resources. The qmaster offers policies in three general classes: ticket policies, urgency policies, and the custom policy. The combination of the priorities assigned by each of the classes of policies is used to produce the total priority for each job. The list of pending jobs is then sorted according to that

total priority. If no active scheduling policies apply to any of the pending jobs, the order of the pending job list will not change. All things being equal, jobs are scheduled in first come, first served order.

What follows is a high-level overview of the available scheduling policies and some general guidance as to when they should be employed.

Ticket Policies

The ticket policies are so named because they use ticket distribution to determine job priorities. The general idea of a ticket policy is that one starts with a predefined number of tickets. Those tickets are then distributed out to the jobs in the pending job list based on the policy configuration. After all the tickets have been handed out, the total number of tickets possessed by each job represents that job's relative priority according to that policy.

The three ticket policies are the share tree (or fair-share) policy, the functional ticket policy, and the override ticket policy. The ticket counts assigned to a job by each policy are combined together to create a final ticket count that represents that job's relative priority according to the ticket policies. (The exact details of how the three ticket policies' ticket contributions are combined together is actually a complex topic well beyond the scope of this paper.)

Share Tree Policy

The share tree policy attempts to assign users and projects a targeted share of the cluster's resources. The share tree policy should be used when fair sharing of resources is desired. A common case for using the share tree policy is when two or more groups pool their resources into a single cluster, but each wants to be assured access to as many resources as the group contributed to the cluster. For example, an administrator might configure the share tree policy to give the QA department 30% of the cluster's resources, as 30 of the 100 servers in the cluster were purchased using budget from the QA department. The user `dant` might be assigned 25% of the QA department's share (or 7.5% of the total cluster's resources) because `dant`'s work takes precedence over the work of the other 8 members of the team who will share the remaining 75% of the QA department's share (or 22.5% of the total cluster's resources).

The share tree policy is named for the way that the target resource shares are defined. Administrators set up target resource shares in a usage (or "share") tree. The top node of the tree represents all of the cluster's resources. At each node that has children (nodes attached to it), the tickets representing the resource share for the node are divided among the node's children (that have pending jobs) according to the child nodes' target shares. The final ticket counts for the leaf nodes of the tree are then distributed to the jobs belonging to those users and projects. The tickets assigned to each job in this way are that job's total share tree ticket count and represent that job's relative priority according to the share tree policy. An example of share tree resource distribution is shown below in Figure 3.
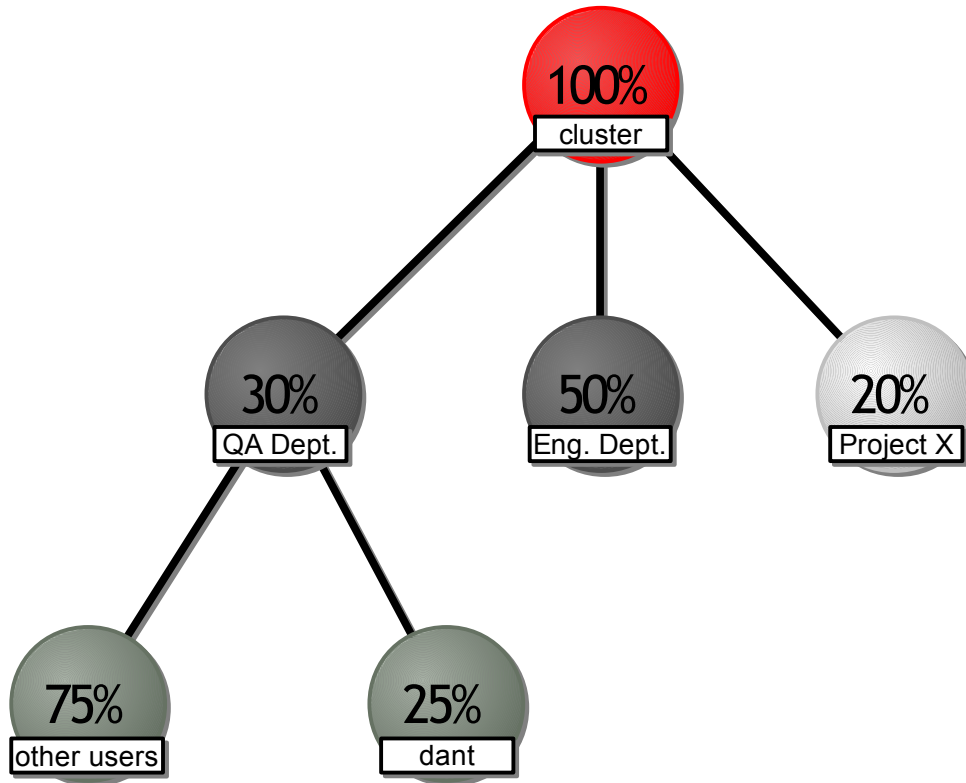
Figure 3: Example Share Tree

An example may help to clarify the process. Assume that the share tree shown in Figure 3 is the share tree configured for the share tree policy. Also assume that the share tree policy is configured to start with 1000 tickets. The user `dant`, from the QA department, has one pending job, and a user working on Project X also has a pending job. The first step is to assign all 1,000 tickets to the root node, the one labeled "cluster." Next, the root node's tickets are divided among its children with pending jobs according to their shares. Both the QA department and Project X have pending jobs, but the engineering department does not. The 1000 tickets from the root node will therefore be divided between just the QA department and Project X, giving the QA department 600 tickets and Project X 400 tickets. (Relative shares are preserved.) Because the Project X node has no children, that node's 400 tickets are not further subdivided. All 400 tickets go to the one pending job for Project X. The QA department node does have children, though, so its 600 tickets are further subdivided. However, only the user `dant` from the QA department has a pending job, so the `dant` node inherits all 600 tickets from the QA department node. Because the `dant` node has no children, the 600 tickets are not further subdivided. All 600 tickets go to the pending job from the user `dant`. In the end, the user `dant`'s pending job receives 50% more tickets than the pending job from Project X, giving the pending job from the user `dant` a higher relative priority.

(In reality the above example is conceptually correct, but is mathematically incorrect according to the actual behavior of the share tree policy. In order to preserve submission order among jobs with equal priority, additional modifiers are applied to the ticket counts for each node, resulting in a somewhat unpredictable ticket count for any given node.)

Another important property of the share tree policy is that it is historical, meaning that past resource consumption is taken into account when calculating target resource shares. In other words, the share tree policy tries to achieve the target resource shares averaged over a period of time, not just at any single point in time. The following graph shows that concept more clearly.
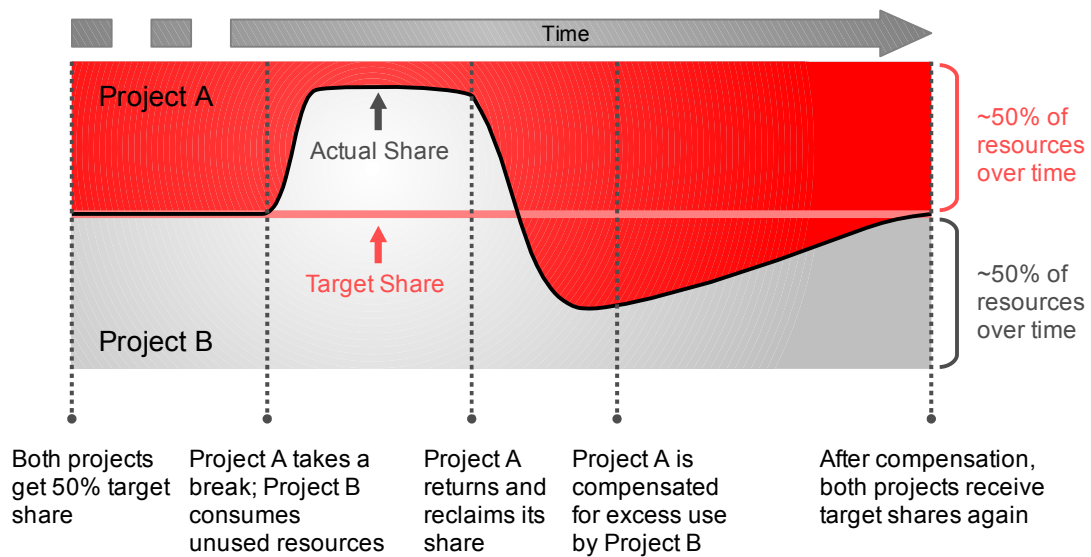

Figure 4: Example of Share Tree Policy Target Resource Share Management

By default, the share tree policy is inactive. The share tree policy should not be used in conjunction with the functional ticket policy as the resulting level of complexity makes scheduling decisions difficult to understand, debug, or tune.

Functional Ticket Policy

The functional ticket policy is similar to a non-historical, single-level share tree policy. It assigns tickets based on four functional categories: users, departments, projects, and jobs. The functional ticket policy should be used when the objective is to set relative priorities among users, departments, projects and/or jobs, but when assigned resource shares are not needed. An example of when the functional ticket policy is useful is an organization with multiple projects of varying priorities that are competing for resources.

Administrators can assign a functional ticket share to any user, department, project, or job to represent the relative ticket share for that user, department, project or job within its functional category, ultimately translating into relative job priorities. If a particular user, department, project, or job has no functional ticket share assigned, it will receive no functional tickets for its category.

For example, the user `dant` has been assigned a functional ticket share of 10. He works in the engineering department, which has been assigned a functional ticket share of 20. The user `andy` has been assigned a functional ticket share of 20, but he works in the QA department, which only has a functional ticket share of 5. If `dant` and `andy` have jobs pending at the same time, `dant`'s job will get half as many tickets in the user category as `andy`'s job (`dant`'s 10 versus `andy`'s 20), but will get four times as many tickets in the department category (The engineering department's 20 versus the QA department's 5), likely resulting in an overall higher ticket count and hence a higher relative priority. Note that because neither job receives tickets in the project or job categories as neither job was assigned a project or job functional ticket share.

After tickets have been assigned to jobs according to the four categories, the ticket assignments are summed for each job to produce a total functional ticket count, which represents that job's relative priority according to the functional ticket policy. (As with the share tree policy example, this example is conceptually correct but leaves out the additional ticket count modifiers used to preserve submission order among jobs of the same priority. These additional modifiers make predicting actual ticket counts extremely difficult.)

Unlike the share tree policy, the functional ticket policy has no memory of past resource usage. All users, departments, projects, and jobs are always assigned tickets purely according to their functional ticket shares. The following graph shows the concept more clearly.
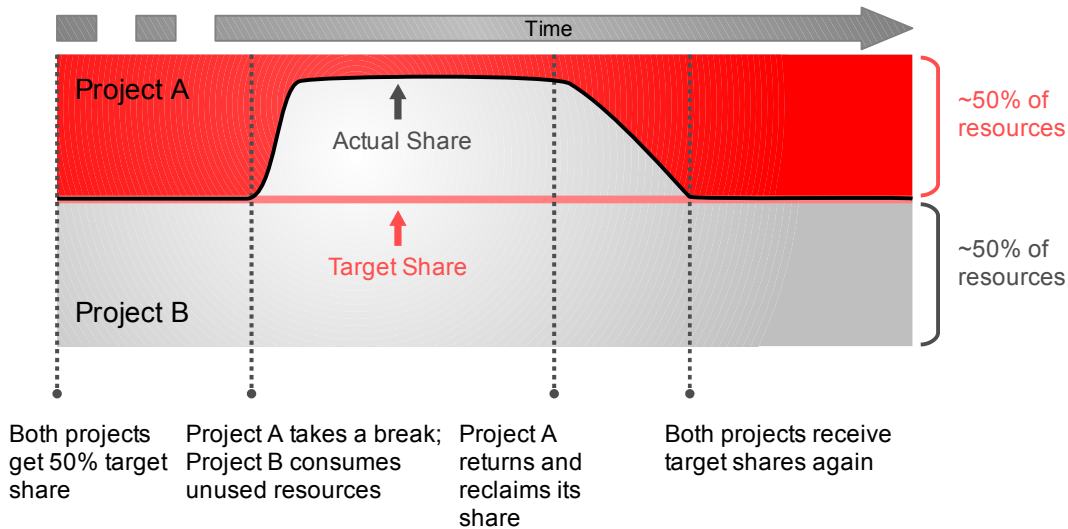

Figure 5: Example of Functional Ticket Policy Target Resource Share Management

By default, the functional ticket policy is inactive. The functional ticket policy should not be used in conjunction with the share tree policy as the resulting level of complexity makes scheduling decisions difficult to understand, debug, or tune.

Override Ticket Policy

The override ticket policy is the simplest ticket policy. Its purpose is to give administrators the ability to temporarily influence the other ticket policies without having to change those policies' configurations. The override ticket policy allows an administrator to assign a user, department, project, or job an arbitrary number of extra tickets. In the case of users, departments, or projects, those tickets are distributed among the pending jobs that belong to them. Tickets can also be assigned directly to a pending job. The override tickets for each job are summed together to give a total override ticket count that represents that job's relative priority according to the override ticket policy.

The override ticket policy is useful whenever either of the other two ticket policies is in use. It makes little sense to use on its own, however, as it is neither as flexible as the functional ticket policy nor as straightforward as the custom policy.

An example use of the override ticket policy might be a project that is running late and needs its remaining work to be completed as soon as possible. An administrator could assign override tickets to that project, effectively boosting that project's jobs' ticket counts and hence their relative priority. When the project's work has been completed, the administrator can then remove the override tickets, returning the cluster to normal.

It is important to remember that the total ticket counts for each job from each of the three ticket policies will be combined together to produce a final total ticket count that represents that job's relative priority. Because the tickets are combined together, the override ticket policy effectively adds extra tickets to the totals produced by the other policies. Because simultaneous use of both the share tree policy and the function ticket policy is highly discouraged, in most cases "the other policies" is really "the other policy," making the override ticket policy even more powerful.

The override ticket policy is active by default, though it has no effect until an administrator makes override ticket assignments.

Urgency Policies

The urgency policies assign priority to jobs based on the jobs' states. The three urgency policies are the wait time policy, the deadline policy, and the resource urgency policy. The urgencies assigned to each job by each of the urgency policies are combined together to create a total job urgency that represents that job's relative priority.

Wait Time Policy

As a job remains in the pending queue, the wait time policy increases the urgency for that job. The urgency for a job in the pending queue increases linearly from the time that it is first submitted until the job is finally scheduled. As the urgency is unbounded, the wait time policy guarantees that a job that remains in the pending list long enough will eventually become the highest priority job in the list.

The wait time policy can be useful for preventing job starvation. If other policies produce large, consistent differences in priority for jobs from different users or groups, the wait time policy can help ensure that the lower priority jobs will eventually get a chance to run. The wait time policy does not

solve the problem of starvation of jobs with large resource requirements, however. To solve that problem, see the sections on resource reservation and advance reservation.

By default, the wait time policy is inactive.

Deadline Policy

One of the properties of a job's definition is the job's deadline. The deadline is the time by which the job must be scheduled. As a job approaches its deadline time, the urgency of that job increases exponentially in an attempt to make sure the job is scheduled in time. A deadline job will reach its maximum urgency at its deadline time, and it will maintain that urgency until it is finally scheduled.

Because the deadline policy has a potential for abuse, the ability to submit deadline jobs is restricted to a limited set of users. In order to submit a job with a deadline, a user must be a member of the `deadlineusers` group. By default, the deadline policy is active, but no users are members of the `deadlineusers` group.

The deadline policy is important for situations where work needs to be processed in a timely manner. Using the deadline policy, jobs with start time or completion deadlines can be guaranteed to move to the front of the pending job list as their deadlines approach. The deadline policy does not solve the problem of starvation of jobs with large resource requirements, however. To solve that problem, see the sections on resource reservation and advance reservation.

By default, the deadline policy is active, but no users are members of the `deadlineusers` group.

Resource Urgency Policy

If some resources in a cluster are particularly valuable, it might be advantageous to make sure those resources stay as busy as possible. A good example is high-cost application licenses. If an organization is paying $20,000 a year for an application license, it is better that it be in use while a $2,000 server is idle than vice versa. To facilitate the preferential scheduling of jobs that use costly resources, the Oracle Grid Engine software offers the resource urgency policy.

Every complex has an urgency value associated with it. For most complexes the urgency value is 0. When a job is submitted with a hard request for a resource, it inherits that resource's urgency value. If a job requests more than one resource or multiples of a single resource, all the urgency values are summed together to produce a total resource urgency for the job.

By default, the only complex with a non-zero urgency value is the slots complex, which is a queue-level consumable that represents available job slots. All jobs implicitly make a hard request for the slots consumable, so what is the point of it having an urgency value? Some jobs, namely parallel jobs, implicitly request multiple job slots. A parallel job that requests four job slots gets four times the urgency of a batch job that, by definition, only requests one job slot. This increased urgency means that if a batch job and a multi-slot parallel job were submitted at exactly the same time, the parallel job would start out at a higher urgency than the batch job and hence would likely be scheduled first. Of course, other resource requests and other policies could affect the jobs' final priorities.

The resource urgency policy is extremely flexible and can be used to solve a wide variety of problems beyond prioritizing use of costly resources. The topic of clever uses for the resource urgency policy is far too broad to cover in this paper.

By default, the resource urgency policy is active, but only the slots complex has a resource urgency assigned.

Custom Policy

The custom policy is so named because it can be used to allow an external entity to dictate priority. It is also known as the *POSIX* policy because it uses a POSIX-standard priority value range and as the *priority* policy because it deals directly with job priorities.

The custom policy is very simple. Every job can be assigned a priority value that can range from -1023 to 1024, the higher the priority value, the more priority that job has. Two jobs with the same priority value have the same priority. By default, the custom policy is the largest contributor to overall job priority, so the priority values assigned to jobs have a very significant impact on their overall priority.

The custom policy has several common uses. Because a job's priority value can be set by the user when submitting the job, users can use the custom policy to implement relative job priorities among their own jobs. Because a job's priority value can be changed after the job has been submitted, external systems can influence a cluster's job scheduling order by adjusting the priority values for pending jobs. Because only administrators can set a job's priority value above 0, the custom policy can be used by an administrator to push a particular job to the front of the pending job list.

By default, the custom policy is active.

Job Scheduling

The second step of the scheduling process is the actual job scheduling, or assignment of a job to a set of free resources. A job is matched to a job slot according to the resources the job has requested (if any) and by the state of the queues and hosts with available job slots. In a default Oracle Grid Engine installation, a job slot is selected for a job in a four-step process:

- **The list of queue instances is filtered according to hard resource requests.** A hard resource request is a request that must be fulfilled. For example, a user might submit a job with a hard request for a Solaris x86 host. Without such a host, the job cannot be run.

- **The remaining list of queue instances is sorted according to soft resource requests.** A soft resource request is a request that need not be fulfilled. For example, a user might submit a job with a soft request for 8 gigabytes of virtual memory. The job may still run on a machine with less virtual memory, but it might take longer to execute. The queue instance list is sorted in order from queue instances that fulfill the most soft resource requests to queue instances that fulfill the least number of soft resource requests.

- **The top tier of the sorted list of queue instances from the previous step is further sorted according to queue sequence number.** Each queue has an administrator-assigned sequence

number that indicates relative scheduling preference. More than one queue is allowed to have the same sequence number. The queue instance list is sorted in order from lowest sequence number to highest sequence number. In a default installation, `all.q` is the only queue, and it has a sequence number of 0. That sequence number of a queue is inherited by all of its queue instances.

- **The top tier of the sorted list of queue instances from the previous step is further sorted according to the host load.** The "load" for a host is calculated using the load formula defined by the `load_formula` scheduler property. In a default installation, a host's "load" is defined as the average number of processes in the OS's run queue over the last five minutes, divided by the number of CPU cores at that host. The queue instance list is sorted in order from least loaded to most loaded.

"Least Loaded"

The Oracle Grid Engine software allows administrators to specify the definition for "least loaded" for each cluster. In a default cluster, "least loaded" is defined as the host with the lowest value for np_load_avg, which is the average number of processes in the OS's run queue over the last five minutes, divided by the number of CPU cores belonging to that execution host, i.e. the five-minute load average normalized against the number of processor cores.

After the last round of sorting, the job is assigned to the queue instance at the front of the list. This process is repeated for every job in the pending list.

Other Scheduling Features

In addition to scheduling policies, the Oracle Grid Engine qmaster offers several other facilities for controlling the manner in which jobs are scheduled. The most important of these facilities are discussed below.

Resource Quota Sets

Resource Quota Sets (RQS) is a very important feature that gives administrators immense control over how cluster resources are consumed. With RQS, administrators can create sets of rules that govern what types of jobs are allowed to use what resources and in what quantity.

Each RQS rule describes a resource limit and to which jobs the limit applies (known as the *filters*). The filters are defined by submitting user or user group, destination host or host group, destination queue, target project, or use of parallel environments, or any combination thereof. For example, an RQS rule might limit jobs submitted by users in the QA department for Project X to no more than 1 job slot per host for hosts in the production host group. (In RQS syntax, such a rule might look like: `limit users @QA projects ProjectX hosts {@Production} to slots=1`. For more information on the RQS syntax, see the `sge_resource_quota`(5) man page[5].)

Rules are aggregated together into rule sets. When evaluating the resource quotas for a job, each rule set is evaluated, and the most restrictive limit produced by any of the rule sets is applied to the job.

---

[5]sge_resource_quota(5) Man Page:
http://gridengine.sunsource.net/nonav/download/man_pages_62u6/htmlman5/sge_request.html?pathrev=V62u6_TAG

Within a rule set, the rules are evaluated one by one, from top to bottom, until the first rule is found that applies to the job in question. That rule then sets the limit for that rule set for that job.

Resource Reservation

When a job requests a large amount of resources, the possibility exists that the job may never run, even though the cluster theoretically offers the required resources. A job may be the highest priority job in the system, but if the entire set of resources needed for the job to run are never all available at once, the job cannot be scheduled. Smaller, lower priority jobs can keep enough of the resources occupied that the large-resource job may stay pending indefinitely.

The solution to this problem is resource reservation. Resource reservation allows the highest priority pending job to gather the resources it needs as they become available, until it eventually has all the resources it needs to run. In order to minimize the impact of reserving resources, the scheduler will *backfill* jobs onto the reserved resources while waiting for the remaining resources to become available. Backfilling will only happen with jobs the scheduler believes will complete before the remaining resources are expected to become available.

Advance Reservation

There are times when it is important to coordinate the availability of cluster resources with external factors such as individuals' schedules or equipment or facility availability. For example, using the cluster for real-time processing of experimental data requires that the compute resources be available at the time of the experiment. Advance reservation allows a user to reserve required resources for a specific time and duration. Once an advance reservation has been granted, the resources are blocked off for that user at the specified times. Users can submit jobs to the reservation, or they can use the reserved resources directly or not at all. In order to minimize the impact of reserving resources, the scheduler will backfill jobs onto the reserved resources while waiting for the start time of the reservation. Backfilling will only happen with jobs the scheduler believes will complete before the reservation's start time.

Exclusive Host Access

Some types of HPC workload are very resource intensive. For example, many HPC workload require large amounts of memory. With such workloads, if two jobs land on the same host, they may exhaust the physical memory available on the host, forcing the host's operating system to swap memory. Swapping memory can introduce a tremendous amount of overhead is such a case, causing both jobs to take longer to complete than the sum of their execution times if there were no memory contention.

Working around this problem in a mixed workload environment can be challenging. Starting with the Oracle Grid Engine 6.2 update 3 release, the software provides the ability to configure a resource as an *exclusive* resource. Any job that requests an exclusive resource will only be scheduled onto a host that is otherwise unoccupied, and once the job has been placed onto a host, no other jobs will be co-scheduled to that host. In the case of a parallel or array job, every task in the job will be scheduled onto a host exclusively. The exception is that if a parallel job's parallel environment is configured to

place more than one slave task on a single host, those co-located slave tasks will be allowed to be co-scheduled onto a host, but no other jobs or job tasks will be scheduled to that host.

Job Submission Verifiers

The Oracle Grid Engine 6.2 update 2 release introduced a new feature called Job Submission Verifiers (JSVs). Strictly speaking, JSVs are not a scheduling feature, but the feature is nonetheless relevant to the scheduling process. A JSV a filtering process configured by an administrator that can accept, reject, or modify incoming job submissions. A JSV is an executable script or binary that adheres to a specific input/output contract. To make JSV development easier, the Oracle Grid Engine product distribution includes API implementations for C, Perl, Tcl, and Borne shell that hide the details of that input/output contract. JSVs can be either configured to execute on the client side, before sending the job submission to the qmaster, or on the server side, before accepting the job submission and placing it in the pending job list.

JSVs can be used in a cluster to enforce policies before the jobs reach the scheduler, improving the scheduling performance and reducing the administrative overhead involved in managing out-of-policy jobs. By rejecting inappropriate jobs at the time of submission, users are immediately made aware of the problem, and the cluster is freed from having to include those jobs in the scheduling process. The ability of JSVs to alternatively make on-the-fly changes to job submissions to correct problems or bring a job into compliance with policies can further simplify the process for uses and administrators.

An important caveat with JSVs is the performance overhead they introduce. While the performance overhead associated with a JSV may be small, when it is run once for every incoming job, the scale of the cluster can cause that small overhead to have a large effect. There are two important considerations when using JSVs. First, place as much as possible into client-side JSVs, where the performance overhead has considerably less impact. Reserve server-side JSVs for validation that is critical and preferably quick. Second, it is highly recommended that writing JSVs in shell scripting languages, such as Borne shell or C shell, be avoided. In such languages, the interpreters launch a large number of processes, each of which adds additional overhead. JSVs written in shell scripting languages should especially be avoided on the server side, where the multiplicative effect can produce huge performance penalties.

Additional Information on Job Scheduling

Job scheduling with the Oracle Grid Engine software is a very large topic. The Oracle Grid Engine software provides a wide variety of scheduling policies with a great degree of flexibility. What is presented above only scratches the surface of what can be done with the Oracle Grid Engine 6.2 software. For a more complete discussion of Oracle Grid Engine job scheduling see the Oracle Grid Engine 6.2 Admin Guide or Scheduler Policies for Job Prioritization in the N1 Grid Engine 6 System[6].

---

[6]Scheduler Policies for Job Prioritization in the N1 Grid Engine 6 System:
http://www.sun.com/blueprints/1005/819-4325.html

Chapter 3: Planning an Oracle Grid Engine Installation

A significant portion of the work involved in installing an Oracle Grid Engine cluster is in preparing the computing environment before installing the software. The Oracle Grid Engine software has some specific needs with respect to file systems, naming services, and user accounts. Once the computing environment has been properly prepared, the actual software installation is fairly simple.

Installation Layout

Before discussing how to configure the computing environment for an Oracle Grid Engine installation, it makes sense to first describe the layout of a typical installation.

The directory in which the Oracle Grid Engine software is unpacked in known as the *root* or `SGE_ROOT` (because the directory path is stored in the `$SGE_ROOT` environment variable). The root directory contains all of the files required to run the Oracle Grid Engine software on one or more binary architectures. The root directory is self-contained, meaning that the installation process does not create or modify files anywhere other than in the root directory.

In order to support multiple binary architectures from the same root directory, the bin and lib directories each have architecture-specific subdirectories. For example, on a Solaris x86 host, the path to the `qsub` command will be `$SGE_ROOT/bin/sol-x86/qsub`.

The installation process will create a new *cell* (or `SGE_CELL`) directory in the root directory. The cell directory contains all of the files associated with the installation. Because the cell directory is completely self-contained, multiple cells can coexist with in the same root directory, with each cell representing a different Oracle Grid Engine installation. As long as the cells do not share port numbers or spool directory paths, all cells under a given root may be simultaneously active, i.e. they may each have a running set of daemons associated with them.

```
$SGE_ROOT/
    bin/
            sol-amd64/
            lx24-ia64/
            ...
    include/
    lib/
            sol-amd64/
            lx24-ia64/
            ...
    man/
    $SGE_CELL/
            common/
                    act_qmaster
                    settings.csh
                    settings.sh
                    sge_request
                    ...
            spool/
                    qmaster/
                            heartbeat
                            messages
                            qmaster.pid
                            ...
                    myhost/
                            active_jobs/
                            execd.pid
                            messages
                            ...
                    ...
    ...
```

Figure 6: Contents of the cell directory

Each cell directory contains a `common` directory and a `spool` directory. The `common` directory contains configuration files, utility scripts, and some cluster data files. The `spool` directory contains some or all of the daemon log (messages) files and may contain the qmaster's data store. For more information on data spooling, see below. Most importantly, the `common` directory inside the cell directory contains the files used by clients to find the qmaster. When a client attempts to contact the qmaster, it reads the `$SGE_ROOT/$SGE_CELL/common/act_qmaster` file to find out on which host the qmaster currently resides. (Because of fail-over, the qmaster may switch hosts periodically.) For this reason, it is important that the cell directory be shared with all potential client hosts, including the execution hosts.

QMaster Data Spooling

The majority of the configuration information for an Oracle Grid Engine cluster, including the complete data for all active jobs, is stored in the qmaster's data spool. The data spool ensures that the state of the cluster is persistent between qmaster restarts. The Oracle Grid Engine software offers three options for the type of data spool as well as the choice of where the data spool should be stored.

The qmaster's data spool can be based on flat files (known as *classic spooling* because it was once the only option), a local Berkeley Database data store, or a remote Berkeley Database server. The computing environment, the size of the cluster, and the expected cluster throughput are key factors used in determining which spooling option is best. Before making any decision based on performance expectations, one should always do performance testing in the local compute environment to determine the best solution.

Some general rules exist that may be helpful in selecting the best-suited data spooling option. For smaller clusters or clusters with less demanding throughput levels, classic spooling offers greater flexibility and ease of administration. Large clusters or clusters with high throughput will see better performance by using a local Berkeley data store. For large or high-throughput clusters with strict high-availability needs, a remote Berkeley Database server may be the best option.

The choice of data spooling options affects one's ability to configure high-availability through Oracle Grid Engine shadow masters. When the qmaster fails over to another node via a shadow master, it must still have access to the data spool in order to recreate the state of the cluster before the previous qmaster failed. A common solution to this problem is to make the data spool available via a shared file system, usually NFS. With local Berkeley spooling, this approach presents a problem as the Berkeley Database software does not work with NFSv2 or NFSv3, and NFSv4 is not yet widely adopted. Depending on the size and throughput of the cluster, classic spooling or a remote Berkeley Database server can be used instead. If shadow masters will not be used in the cluster, the qmaster's data spool need not be shared. In fact, in a cluster with no shadow masters, it is possible to configure the cluster with no shared file system at all.

The choice of data spooling options also affects where the data spool is located. For classic spooling, the data spool is located in the qmaster's *spool* directory, which is located at `$SGE_ROOT/$SGE_CELL/spool/qmaster` by default. As explained above, however, a local Berkeley Database data store often cannot be located on an NFS-shared file system. As mentioned in the previous section the cell directory is usually NFS-shared. To resolve this conflict, the qmaster allows the local Berkeley data store to be located in a directory that is different from the qmaster's spool directory.

Execution Daemon Data Spooling

Each execution daemon has its own spool directory. The default location for the spool directory offered by the installation process is `$SGE_ROOT/$SGE_CELL/spool/<hostname>`. Because the cell directory is usually NFS-shared, that means that every execution daemon's spool directory is also NFS-mounted. No process needs access to the execution daemon's spool directory aside from the execution daemon, so it is almost always a good idea to configure every execution daemon with a

"local" spool directory, meaning that the spool directory is located on a file system that is not NFS-mounted.

The execution daemon's spool directory contains information about the jobs currently being executed by that execution daemon, so that in the event of a failure, the execution daemon can try to reconstruct its previous state when it restarts. Among the information stored in the execution daemon's spool directory is a local copy of every job script being executed by that execution daemon. When a user submits a binary job, only the path to the binary is passed on to the execution daemon. When a user submits a script job, however, the script file is copied and sent along with the job. When the execution daemon executes the job, it runs a local copy of the script saved into its spool directory, rather than the original script file submitted by the user.

Managing User Data

The Oracle Grid Engine software does not manage user data by default. Whatever data is needed by users' jobs must already be accessible from the execution hosts, normally via some kind of shared or distributed file system, such as NFS or the Lustre™[7] file system. It is very important to plan effectively for the handling of user data. Poorly planned data management can bring even the best cluster to its knees.

A shared or distributed file system is the most common approach to data availability. The advantage of a shared or distributed file system is that every job sees the same set of data, and there are no data synchronization issues.

Shared and distributed file systems introduce their own set of problems. The quality of service of an NFS file system tends to decay rapidly as the data throughput increases. Lustre, on the other hand, has excellent scalability characteristics, but is a heavyweight solution that requires a significant amount of administrative attention. There are other alternatives, such as QFS or pNFS, and each comes with its own set of advantages and limitations. If a shared or distributed file system is chosen as the approach to data availability, be sure to examine all the options and pick one that is appropriate for the computing environment and the needs of the jobs.

Instead of using a shared or distributed file system, the Oracle Grid Engine software offers some alternatives to making user data available at the execution hosts. The first and most direct is the use of a *prolog* and *epilog* for data staging. A prolog is an executable configured on the host or queue to be run before each job executed in that queue or on that host. An epilog is the same, except that it runs after each job. A prolog and epilog can be configured to read the data profile for a job from the job's environment variables and stage the information in and/or the results out.

Data staging with a prolog and epilog works well for moderately sized data sets. For jobs that need very large amounts of data, the data transfer can take a significant amount of time. During that time, the job is not yet able to execute because it doesn't have its data, but it is nonetheless occupying a job slot on the execution host. For jobs with large data needs, an alternative approach to data staging is to

---

[7]Lustre Product Page:
http://www.oracle.com/us/products/servers-storage/storage/storage-software/031855.htm

submit a separate data staging job(s). Using job dependencies (also known as workflows), a user can submit a stage-out job that depends on the actual job that depends on a stage-in job.

For data sets that may be used by more than one job, either of the above solutions can be amended by adding a complex that tracks the availability of the data set on each machine. Jobs can then be submitted with a soft request for that resource. If a host is available with the data set already loaded, the job will go there. If not, it will go to a host without the data set loaded and load it there before executing the job.

Naming Services

The Oracle Grid Engine software is relatively agnostic to the naming service used in the compute environment. The most important thing is that the master host be able to resolve the host names of all the execution hosts and client hosts, and that all client hosts and execution hosts be able to resolve the name of the master host. Two host name resolution configuration options are provided to simplify matters even further.

During the installation process, the installing user will be asked if all of the hosts in the cluster share the same domain name. If the user answers 'yes,' the Oracle Grid Engine software will ignore domain names completely by truncating every host name at the first dot ('.'). If the user answers 'no,' the Oracle Grid Engine software will always use fully qualified host names.

After the installation process is complete, the administrator has the option to set up a host aliases file. Using the host aliases file, it is possible to force the Oracle Grid Engine software to always map one host name to another. For example, if an execution host sets its host name as `hostA`, but the master host resolves that execution host's IP address as `hostB,` the host aliases file can be used to always map `hostA` to `hostB`. For more information, see the `host_aliases`(5)[8] man page.

User Accounts

When an execution daemon executes a job, it executes the job as the user who submitted the job. To enable that functionality, the name of the user who submitted the job is included in the job's definition. It is important to note that it is the name of the user who submitted the job, not the id, that is passed to the execution daemon. In order to execute the job as that user, that user's name must exist as a user account on the execution host. The execution daemon relies on the user management facilities of the underlying OS to resolve the user name to a local account. How the user name is resolved, be it through `/etc/passwd` or LDAP or some other mechanism, plays no role, as long as the user name can be resolved to a local account.

Admin Accounts

---

[8]host_aliases(5) Man Page:
http://gridengine.sunsource.net/nonav/download/man_pages_62u6/htmlman5/host_aliases.html?pathrev=V62u6_TAG

To enable the use of a privileged port and the ability to execute jobs as other users, the Oracle Grid Engine daemons must be started as the root user. It is actually possible to install and run a cluster as a non-root user, but then only that user will be able to run jobs in the cluster. To limit the security impact of the daemons running as root, they instead switch to a non-root user immediately after start-up. That non-root user can be specified during installation, and is by convention chosen to be `sgeadmin`. If the cluster is configured to use a non-root user in this way, it is important that the user chosen have an account on the master machine and every execution host.

Service Ports

An Oracle Grid Engine cluster requires two port numbers in order to function: one for the qmaster, and one for the execution daemons. Those port numbers can be assigned in one of two places, chosen during the cluster installation process. One option is to read the port numbers from the `sge_qmaster` and `sge_execd` entries in the `/etc/services` file. The other is to set the port numbers in the shells of client users.

It is important to note that before interacting with an Oracle Grid Engine cluster, every client must setup his or her environment. For Borne shell (and derivative) users, that means executing the command, ". `$SGE_ROOT/$SGE_CELL/common/settings.sh`". For C shell (and derivative) users, that means executing the command, "`source $SGE_ROOT/$SGE_CELL/common/settings.csh`". Executing that command imports a set of environment variables into the user's shell. Among the environment variables imported are variables that contain the port number of the qmaster and execution daemon in the case that they are not to be read from the `/etc/services` file.

It is usually more convenient not to read the port numbers from the `/etc/services` file for two reasons. First, reading the port numbers from the `/etc/services` file would mean adding the port numbers to the `/etc/services` file on every host in the cluster. Second, if all cells read their port numbers from the (same) `/etc/services` file, all cells must, by definition, be using the same port numbers, which means that only one of the cells may be running at a time. Reading the port number from the user's shell means that every cell can use different port numbers and hence can be running simultaneously. Whether or not running more than one cluster on the same set of hosts is a good idea or not is a different discussion.

Oracle Grid Engine port numbers

Oracle Grid Engine clusters will commonly be found using three well-known sets of port numbers. The first pair is 6444 and 6445 (for the qmaster and execution daemons, respectively). Those port numbers are assigned by IANA to Oracle Grid Engine. The second pair is 575 and 576. Those port numbers were in common use during the 6.0 time frame, before the IANA port numbers were assigned. The last pair is 535 and 536. Those port numbers were popular in the 5.3 time frame and earlier. The best practice is to use the IANA port numbers for all new clusters. It is still common, however, to see clusters using one of the older sets of port numbers, either for historical reasons or because the installing administrator hasn't yet adopted the new port numbers.

Chapter 4: Suggested Oracle Grid Engine Configurations

The Oracle Grid Engine software provides administrators with a tremendous amount of flexibility in configuring the cluster to meet their needs. Unfortunately, with so much flexibility, it sometimes can be challenging to know just what configuration options are best for any given set of needs. What follows is a series of common cluster configurations based on various cluster parameters. It is likely that no one cluster configuration described below will meet all the needs of any particular compute environment, but by borrowing ideas and suggestions from one or more of the suggested configurations, it should be possible to arrive at a reasonable cluster configuration without a tremendous amount of research and planning.

General Suggestions

Some cluster configurations apply to clusters of any size. The following recommendations apply universally:

- It is almost always a good idea to have the execution daemons use a local spool directory. Having the execution daemon spool directories located on an NFS-mounted file system can potentially have troubleshooting benefits, but only in the smallest clusters do these benefits outweigh the cost.

- To give users the flexibility to increase their job priorities, it is almost always a good idea to configure the default priority for all users' jobs to be less than 0. Users cannot submit jobs with a priority greater than 0. The default priority, however, is already 0, which means that users can only reduce the priorities of their jobs but not increase them. Because users will rarely volunteer to reduce the priority of a job, setting the default priority to a value less than 0 gives users some room to increase the priority of jobs they feel are important. -100 is a suggested value. To set the default priority value, see the -p option in the qsub(1) man page[9] and the sge_request(5) man page[10].

- Configure the scheduler for immediate scheduling by setting the flush_submit_sec and flush_finish_sec scheduling parameters to 1. This value is the number of seconds to wait after a job is submitted or a job ends (respectively) before triggering a scheduler run. These parameters were historically useful before multi-core machines became pervasive. The idea of inserting a delay was to aggregate job submissions and handle them in a single bundle, rather than constantly interrupting other cluster functions to run the scheduler. On a machine with at least two processor cores, however, there is no penalty to having the scheduler run as often as there is work to be done. As the scheduler is currently only a single thread, it will consume only a single processor core, leaving the other processor core available to execute the rest of the cluster functions. In the

---

[9] qsub(1) Man Page:
http://gridengine.sunsource.net/nonav/download/man_pages_62u6/htmlman1/qsub.html?
pathrev=V62u6_TAG

[10] sge_request(5) Man Page:
http://gridengine.sunsource.net/nonav/download/man_pages_62u6/htmlman5/sge_request.html?
pathrev=V62u6_TAG

case that the master machine has only a single processor core, follow the advice in the individual sections below.

- Set the scheduling interval to 20 seconds. Because immediate scheduling is enabled by setting `flush_submit_sec` and `flush_finish_sec`, the scheduling interval only governs the freshness of the scheduling information in most cases. The scheduling interval does, however, serve as the absolute upper bound on how long a job could wait before the scheduler attempts schedule it, and may in some cases serve additional roles. In the case that the master machine has only a single processor core, follow the advice in the individual sections below.

Small Clusters

Small Cluster, Low Throughput

In Oracle Grid Engine terms, a small cluster is one with roughly 64 or fewer hosts. The Oracle Grid Engine software does not measure scalability in terms of CPU cores. A Sun SPARC Enterprise T5240 Server with 128 threads creates no more burden on the qmaster than a single-socket, single-core blade server, aside from the obvious fact that the qmaster will have to supply a 128-thread server with new jobs at a faster rate to keep it busy than would be needed for a single-core server.

The other measure of an Oracle Grid Engine cluster is job throughput: how many jobs are processed by the system in a given unit of time. Low throughput for an Oracle Grid Engine cluster is anything less than about a job per minute or around 1,500 jobs a day.

For a small, low-throughput cluster that processes mainly batch and array jobs, the following recommendations apply:

- Use classic spooling as the spooling method. Classic spooling will provide greater flexibility, and with a small, low-throughput cluster, it will have no noticeable performance penalty.

- Configure the scheduling interval to be somewhere between 4 to 8 seconds. With immediate scheduling enabled as described above, the scheduling interval has very little effect on the scheduling of jobs. It does, however, control the freshness of the information reported by the scheduler. Reducing the scheduling interval gives users a more up-to-date picture of the state of the pending jobs.

- In some cases, it may be appropriate to set the scheduling interval to 1 second, meaning that the scheduler will be running almost constantly. As long as the master machine can handle the increased load, this approach can result in excellent end-user performance characteristics. If the scheduling interval is set to 1, `flush_submit_sec` and `flush_finish_sec` no longer need to be set.

In a small, low-throughput cluster, the easiest approach is often to share the root directory via NFS. Doing so makes all of the application binaries for all architectures automatically available for all hosts, including clients. It also means that the cell directories are all automatically shared. In such a cluster, the performance impact should not be significant. It is nevertheless generally a good idea not to have both the shared data for jobs and the root directory shared from the same NFS server.

- Starting with Oracle Grid Engine 6.2, the `schedd_job_info` scheduler setting is set to false by default as having it set to true can cause performance issues in large clusters. Having `schedd_job_info` set to false, however, disables the ability of `qstat -j <jobid>` to report why a job is still in a pending state. As this information is extremely useful for end users, it is suggested in small, low-throughput clusters that `schedd_job_info` be set back to true, with the caveat that administrators should periodically check on the qmaster's memory consumption, to make sure `schedd_job_info` isn't causing problems.

Small Cluster, Mixed Batch/Parallel Workload

For a small, low-throughput clusters that process mainly batch and array jobs, the following recommendations apply in addition to the recommendations from the preceding section:

- In order for a cluster to support parallel jobs, an administrator must configure one or more parallel environments. A parallel environment configuration tells Oracle Grid Engine how to start, stop, and manage a parallel message-passing framework that is used by parallel jobs. For information about configuring parallel environments in general, see the Oracle Grid Engine 6.2 Administration Guide and the `sge_pe(5)` man page[11]. For help configuring parallel environments for specific frameworks, see the *Howto* pages[12] on the Grid Engine open source project site[13].

- Because a resource-heavy parallel job can be starved by a series of lower-priority batch jobs or smaller parallel jobs, it is recommended that resource reservation be enabled by setting the `max_reservation` scheduler setting to a value greater than 0. The value of `max_reservation` is the number of resource reservations that will be honored during a scheduler run. If `max_reservation` is set to 2, each scheduler run the two highest priority jobs with resource reservations will have their reservations honored. See the Resource Reservation section in Chapter 2 of this paper for more information.

- The purpose of the Oracle Grid Engine software is to distribute workload as efficiently as possible. In most cases, that means spreading the workload evenly across all of the machines in the cluster. That strategy works well for batch jobs, but when the workload consists of a mixture of batch jobs and shared-memory parallel jobs, it can cause problems.

A shared-memory parallel job must run all of its job tasks on the same host in order to communicate among them using a shared memory region. Imagine a cluster with four 8-core hosts already executing eight batch jobs. The scheduler would naturally have scheduled those eight batch jobs one per host. Now imagine that a user submits an 8-way shared-memory parallel job. Even though there are still 28 job slots available in the cluster, the shared-memory job cannot run because it needs eight slots all on one machine.

---

[11] sge_pe(5) Man Page:
http://gridengine.sunsource.net/nonav/download/man_pages_62u6/htmlman5/sge_pe.html?
pathrev=V62u6_TAG

[12] Grid Engine Howtos Page: http://gridengine.sunsource.net/howto/howto.html

[13] Grid Engine Open Source Site: http://gridengine.sunsource.net/

The solution for clusters that host shared-memory parallel jobs is to segregate shared-memory jobs from all other jobs, as long as there are no resource conflicts. See Appendix A for details about how to configure a cluster in this fashion.

Mid-sized Clusters

Mid-Sized Cluster, High Throughput

A mid-sized Oracle Grid Engine cluster is anywhere from around 64 hosts up to about 256 hosts. A high-throughput cluster is one that sees more than about 100 jobs per minute or hundreds of thousands of jobs per day. For a mid-sized, high-throughput cluster, the following recommendations apply:

- Because of the danger of performance problems, the `schedd_job_info` scheduler setting should be set to false in a high-throughput cluster. With the currently available versions of the Oracle Grid Engine software, setting `schedd_job_info` to false will make information unavailable to users about why jobs are still pending. With a later release, a new means will be added to make that information available to users.

- In a high-throughput cluster **with a single-core master host**, the `flush_submit_sec` and `flush_finish_sec` scheduler settings should be set to 4 or more seconds, depending just how high the throughput is. At the same time, the scheduling interval should be set to 3 minutes to reduce unnecessary scheduler overhead.

- In a mid-sized cluster without shadow masters, the qmaster's data spool should be a local Berkeley data spool. In larger clusters, the local Berkeley data spool performs better than classic spooling. In particular, in high-throughput clusters classic spooling could run out of inodes. Local Berkeley spooling has no such issue.

- Sharing the entire root directory for a mid-sized cluster may reduce performance. The cost in additional overhead from increased NFS traffic could begin to outweigh the ease of management afforded by a shared root directory, especially in clusters that also run parallel jobs. In any case, the cell directories themselves should still be shared.

- In a high-throughput cluster, repeated execution of the `qstat` command by users can place significant burden on the qmaster. In such a cluster it is a recommended best practice to either restrict access to the `qstat` command to administrative users or to create a `qstat` wrapper that returns cluster status information from a cache that is updated only periodically.

- A high-throughput cluster will generate a large amount of reporting data. The Oracle Grid Engine software has two outlets for reporting data. One is the *accounting file* that is read by the `qacct` command, and the other is the *reporting file* that is read by the Accounting & Reporting Console (ARCo).

When a users issues a `qacct` command, `qacct` will read the accounting file line by line, until it has gathered the requested information. In a high-throughput cluster, depending on the frequency with

which the accounting file is rotated, the accounting file could be very large, and the `qacct` command could take tens of seconds to complete.

ARCo works very differently. Periodically ARCo will read and remove the contents of the reporting file and store them in a relational database. When a user runs a query using ARCo's web-based user interface, the information is retrieved from that database, resulting in a much faster response time.

For high-throughput clusters, it is highly recommended that ARCo be established as the preferred mechanism for retrieving historical reporting data, as opposed to the `qacct` command.

- In larger, more active clusters, the various log and reporting files can grow quite large. It is a recommended best practice to establish a file rotation schedule to keep file sizes manageable. The files that may need to be rotated are:

  - $SGE_ROOT/$SGE_CELL/common/reporting

  - $SGE_ROOT/$SGE_CELL/common/accounting

  - <qmaster spool>`/messages`

  - <execd_spool>`/messages`

If the Accounting & Reporting Console is active, it will manage the reporting file automatically.

Mid-Sized Cluster, Secure and Highly Available

By default the Oracle Grid Engine installation process installs in an insecure mode. A Certificate Security Protocol (CSP) secure installation option is available, however, that uses secure certificates to verify user identity and to encrypt all data communications. In order to install using CSP, use the `-csp` switch with the installation scripts. You can find more information in the Installing Security Features chapter of the Oracle Grid Engine Installation Guide.

For end users and administrators, a CSP cluster is identical to an insecure cluster with one exception. A CSP cluster has an additional administrative requirement. For each user of the cluster, including the root user and the admin user, a certificate and private key must be created in the master host's keystore directory (usually in `/var/sgeCA`) and then copied into each user's `$HOME/.sge` directory by using the `$SGE_ROOT/util/sgeCA/sge_ca -copy` command. If for any reason a user's certificates change, they must be updated in both places.

As described in Chapter 1, the shadow master is the Oracle Grid Engine software's built-in high-availability solution. By configuring one or more shadow masters on hosts other than the master host, unplanned downtime can be minimized. Should the qmaster daemon or master host fail, one of the shadow hosts will start a new qmaster instance. When configuring shadow masters for a cluster, the following recommendations should be considered:

- In a highly- available cluster, the cell directory must be shared. When a client or execution daemon attempts to connect to the qmaster, it first looks in the `$SGE_ROOT/ $SGE_CELL/common/act_qmaster` file to see which host is currently the master host. For this reason, all hosts in the cluster must have access to the cell directory.

- Shadow daemons in an HA cluster must also have access to the heartbeat file in the qmaster spool directory. Recall that depending on the spooling method, the spool directory may or may not actually house the data spool. The recommended best practice, regardless of the spooling method used, is to place the qmaster's spool directory in the default location, inside the cell directory. The cell directory must be shared, as explained in the previous bullet, so placing the spool directory in the cell directory means that it will be shared as well.

- When installing a shadow master, the installation process does not offer the option to set the frequency with which the shadow master will check the qmaster's heartbeat file (60 seconds by default) or how long the shadow master will wait after discovering that the qmaster is no longer active before starting a new qmaster (5 minutes by default). To change those parameters, edit the `$SGE_ROOT/$SGE_CELL/common/sgemaster` script to set the `$SGE_CHECK_INTERVAL` and `$SGE_GET_ACTIVE_INTERVAL` variables appropriately. See the `sge_shadowd`(8) man page[14] for more details.

- When installing a cluster with shadow masters, there are three spooling options: classic spooling over NFS, Berkeley spooling over NFSv4, or a remote Berkeley server. Classic spooling over NFS is in most cases the best solution for a highly available cluster. Because the cell directory must be shared, no additional administrative work is required to enable classic spooling over NFS. If an HA NFS service is in place (which is a recommended best practice for a highly available cluster), classic spooling is automatically protected. Classic spooling in a regular cluster has poorer performance than Berkeley spooling, but in an HA environment, the performance cost is less significant. In very large or very high throughput clusters, the performance characteristics of classic spooling may become an issue, at which point a remote Berkeley server may be a better solution.

A Berkeley data spool over NFSv4 is often not an option as NFSv4 is not yet a widely adopted standard. (The Berkeley Database software does not work with earlier NFS versions.) Performance of a Berkeley spool is also significantly degraded when running over NFS.

Because the cell directory must be shared in any case, using a remote Berkeley server is only recommended in special cases, such as clusters with tens or hundreds of thousands of simultaneously active jobs. The goal of installing a shadow master is to eliminate a single point of failure in the cluster, but adding a remote Berkeley server for spooling adds an additional single point of failure.

Large Clusters

Large Cluster, Medium Throughput

A large Oracle Grid Engine cluster is one with more than 256 hosts. (The largest Oracle Grid Engine clusters in production have several thousand hosts.) A medium-throughput cluster is one that processes around 10 jobs per minute, or around 15,000 jobs a day. For a large, medium-throughput cluster, the following recommendations apply:

---

[14] sge_shadowd(8) Man Page:
http://gridengine.sunsource.net/nonav/download/man_pages_62u6/htmlman8/sge_shadowd.html?pathrev=V62u6_TAG

- Because of the danger of performance problems, the `schedd_job_info` scheduler setting should be set to false in a medium-throughput cluster. With the currently available versions of the Oracle Grid Engine software, setting `schedd_job_info` to false will make information unavailable to users about why jobs are still pending. With a later release, a new means will be added to make that information available to users.

- In a medium-throughput cluster **with a single-core master host**, the `flush_submit_sec` and `flush_finish_sec` scheduler settings should be set to 4 to 10 seconds. At the same time, the scheduling interval should be set to 2 minutes to reduce unnecessary scheduler overhead.

- In a large-sized cluster without shadow masters, the qmaster's data spool should be a local Berkeley data spool. In large clusters, the local Berkeley data spool performs better than classic spooling. In particular, in large clusters classic spooling could run out of inodes. Local Berkeley spooling has no such issue.

- Sharing the entire root directory for a large cluster will reduce performance. The cost in additional overhead from increased NFS traffic will likely outweigh the ease of management afforded by a shared root directory. In any case, the cell directories themselves should still be shared.

- Every execution daemon in a cluster reports its current status to the qmaster at a predetermined interval, called the *load report interval.* By default, the load report interval is 40 seconds. In a large cluster, the default load report interval may cause excessive overhead as the qmaster processes the incoming reports. Depending on the size of the cluster, a load report interval from 1 to 5 minutes may be appropriate. Increasing the load report interval increases the amount of time that load report data is allowed to become stale before being refreshed. Increasing the staleness of the load report data is obviously not a desirable side-effect, but it is often an acceptable trade-off for not overwhelming the qmaster with too many load reports.

Multi-Cluster

New with the Oracle Grid Engine 6.2 release is the ability to configure multiple cooperating clusters that are able to share resources using the Service Domain Manager software. For many situations the enterprise cluster functionality of the Oracle Grid Engine software (like fair-share scheduling) is sufficient to allow multiple clusters to be coalesced into one. In some cases, however, the individual clusters must retain their independence, but would nonetheless profit from being able to share resources among the individual clusters. In those cases, the Service Domain Manager software included with the Oracle Grid Engine 6.2 release may be the best solution.

Simply put, the Service Domain Manager migrates resources from clusters where they are under-utilized into clusters where they are needed. The Service Domain Manager software is driven by service-level objectives (SLOs) that define when a cluster is in need of additional resources.

The following recommendations apply to Oracle Grid Engine clusters that are to participate in a Service Domain Manager managed multi-cluster and to the Service Domain Manager configuration itself:

- The Service Domain Manager software does not currently have a notion of leasing resources. Once a resource has been migrated to another cluster, it belongs to that new cluster. In order to implement a form of resource leasing, every resource should be assigned a property that labels the resource according to its original cluster membership. Each cluster's Service Domain Manager agent should define a `PermanentRequestSLO` with a low urgency that requests resources labeled as belonging to that cluster. Such an SLO ensures that as borrowed resources are no longer needed in other clusters, they will be returned to their "home" clusters.

- For resources that are not geographically co-located, the recommended best practice is to assign each resource a property that labels the resource according to its physical location. The Service Domain Manager agents for each cluster should then translate that property into an Oracle Grid Engine complex. Users with work that is location-sensitive can then submit their jobs with a hard request for resources in the appropriate physical location. The same technique can be applied to resolve network differences.

Conclusion

The Oracle Grid Engine software is a very flexible tool for increasing resource utilization and overall job throughput while managing resource usage according to business policies and goals. It is capable of scaling from the smallest departmental clusters all the way up to the enormous Ranger cluster at the Texas Advanced Computing Center, which boasts more than sixty-two thousand processor cores and over one hundred terabytes of aggregate memory.

It is also worthwhile to note that the Oracle Grid Engine software is based on the Grid Engine open source project. The Grid Engine open source project has a thriving community of users and developers who are working with the software on a daily basis. For questions about installation and configuration not covered in the Oracle Grid Engine documentation, the Grid Engine open source community can be an invaluable resource. The community mailing lists, wiki, blogs, and howtos cover a broad range of useful topics. See the Grid Engine open source project site for more information.

This paper has focused on the core concepts behind the Oracle Grid Engine software. It has, however, only briefly touched on several of the other modules included in the Oracle Grid Engine software, such as the Accounting and Reporting Console (ARCo) and the Service Domain Manager (SDM) software. These additional modules each represent deeper topics unto themselves, and hence merit fuller treatment than could be given them in this paper. Look for future papers to address these topics more fully and give them appropriate treatment.

About the Author

Daniel Templeton Product Manager for the Oracle Grid Engine product. Formerly a developer on the Oracle Grid Engine and Service Domain Manager products, Daniel's main contributions were to the Oracle Grid Engine DRMAA implementations and the architecture and design of the Service Domain Manager product. In his current role, he acts as both a customer and community advocate, working to drive interest, adoption, and customer satisfaction for the Oracle Grid Engine and Service Domain Manager products.

Acknowledgments

Daniel would like to thank Miha Ahronovitz, Jeff Beadles, Chris Dagdigian, Roland Dittel, Andreas Haas, Sandra Konta, Lubomir Petrik, Thomas Reuter, and Andy Schwierskott for their feedback and contributions to this paper.

Appendix A: Configuring an SMP/Batch Cluster

Shared-memory parallel jobs execute in multiple job slots on a single host. They do so in order to use a shared memory region for interprocess communications. Because of the way that the Oracle Grid Engine scheduler distributes workload, shared-memory parallel jobs can have scheduling conflicts with regular batch and parallel jobs. To prevent these conflicts, the cluster can be configured to segregate shared-memory parallel jobs from the other jobs in the cluster as long as there are free resources. To configure a cluster in this fashion, follow these steps:

- Step 1: Create a queue for shared-memory jobs that is separate from the queue(s) for other jobs.

- Step 2: Change the `queue_sort_method` scheduler setting from `load` to `seqno`. The default setting of `load` causes the scheduler to behave as described in the Job Scheduling section in Chapter 2. Changing the sort method to `seqno` reverses the order of steps 3 and 4 from the Job Scheduling section. Having steps 3 and 4 reversed means that the scheduler will first select a queue by sequence number and then select the host from that queue that is least loaded.

- Step 3: Assign sequence numbers to the queue instances for the shared memory queue in monotonically increasing fashion, e.g. `shared.q@host1:seq_no=0,` `shared.q@host2:seq_no=1, shared.q@host3:seq_no=2,` etc.

- Step 4: Assign sequence numbers to the queue instances for the other queues in monotonically decreasing fashion, e.g. `all.q@host1:seq_no=50, all.q@host2:seq_no=49,` `all.q@host3:seq_no=48,` etc.

The effect of this configuration is to force the scheduler to place shared-memory jobs first on `host1`, and when `host1` is full, then on `host2`, etc. The scheduler will also be forced to place all other jobs first on `host51`, and when `host51` is full, then on `host50`, etc. This dichotomy means that shared-memory jobs will only conflict with other jobs when all the hosts are occupied. A less rigid solution would be to create two or more host groups and to use the above sequence numbering scheme with those host groups instead of with individual hosts. In that way, the scheduler is still allowed to pick the least loaded host from each host group, instead of demanding that hosts are filled in numerical order by sequence numbers.

Appendix B: Glossary

BATCH JOB

A single segment of work. See *job*.

CELL DIRECTORY

A directory under the root installation directory that contains all of the information about a cluster.

CLUSTER

A group of hosts connected together by the Oracle Grid Engine software.

COMMON DIRECTORY

A directory in the cell directory that contains a variety of data and scripts that are useful to many components in a cluster.

COMPLEX

A resource modeled in a cluster.

COMPUTE CLUSTER

A cluster that is focused on providing compute resources. See *cluster*.

CONSUMABLE

A complex whose value is decremented when jobs that request it are scheduled. See *complex*.

EXECUTION DAEMON

The component responsible for executing jobs using compute resources.

EXECUTION HOST

A host on which an execution daemon is running.

INTERACTIVE JOB.

Cluster-managed interactive login. See *Job*.

JOB

A segment of work to be executed by the cluster, defined as anything executable from a command terminal.

JOB SLOTS

The capacity of a queue to run a job. The number of job slots available on a machine often correlates to the number of processor cores.

LOAD SENSOR

An administrator-defined executable that measures and reports the value of one or more complexes. See *complex*.

MASTER HOST

The host on which the qmaster process is running.

PARALLEL JOB

A job composed of cooperating distributed tasks. See *job*.

PARAMETRIC OR ARRAY JOB

A group of similar work segments operating on different data. See *job*.

QMASTER

The central component of a cluster.

RESOURCE

See *complex*.

SCHEDULING INTERVAL

The period between successive scheduling runs. See *scheduling run*.

SCHEDULING RUN

A single execution of the scheduler in which the entire list of pending jobs is resorted according to policies, and available job slots are filled with jobs from the top of the sorted list.

SHADOW DAEMON

A daemon that manages `qmaster` process failover.

SHADOW MASTER

See *shadow daemon*.

ORACLE®

White Paper Title
[Month] 2010
Author: [OPTIONAL]
Contributing Authors: [OPTIONAL]

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Oracle is committed to developing practices and products that help protect the environment