Batch Share Management Tool

October 2014

Author:
Ties de Kock

Supervisors:
Jérôme Belleman
Ulrich Schwickerath

CERN openlab Summer Student Report 2014

CERN openlab

Contents

1 Project goals

One of the key computing services at *CERN* is the *central batch system*. The central batch service currently consists of around 4000 machines with thousands of users, divided over more than 250 groups.

The central batch service system runs *IBM/Platform LSF®*. Additionally, a pilot service based on *HTCondor* is currently being set up.

In a batch system the worker nodes are shared among the users over time. The central batch service uses *fair-share* scheduling, which ensures that, over a historic time window, a group of users can use the capacity (*share*) that is assigned to them. Each share on the system is allocated to a group of users. Compute coordinators can delegate capacity, as well as the control of this capacity to subgroups. This situation is shown in figure 1.

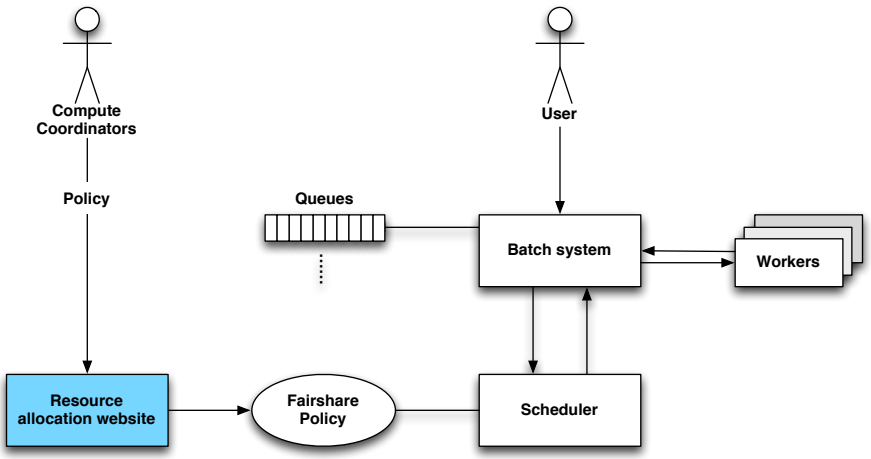

Figure 1: Central Batch System at CERN

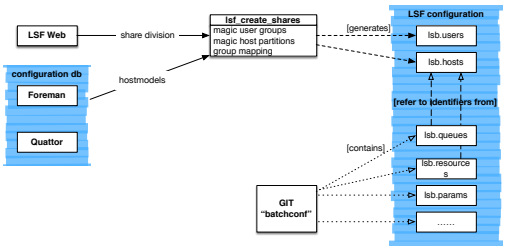

Figure 2: LSF configuration

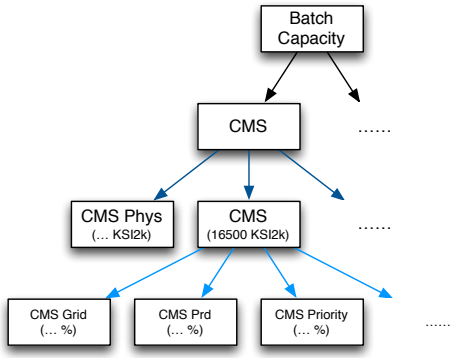

Figure 3: Tree of shares in LSF Web

1.1 Delegated resource allocation

The users want to adjust the capacity assigned to groups, group structure and group memberships. Adjusting the system configuration to match this information manually is a full-time job. This motivated the creation of a resource allocation web application, where the allocation of capacity is delegated to the *compute coordinators*.

Once a day the batch system configuration is updated to match the share database. The sources of information for configuring *LSF* are shown in figure 2.

1.1.1 LSF Web

At the moment *LSF Web* is used to configure the shares. It was developed in 2008. In LSF Web there are three layers of shares: *Accounts*, *Groups* and *Subgroups*. Each of these layers has its own database table (see appendix A.1). Figure 3 shows a part of the allocation tree from LSF Web.

Since 2008, the environment for the system changed. Three major areas need improvement:

Single Sign-On The login system used by LSF Web has been deprecated. The new system should use the new *single sign-on* system.

Multiple resources LSF Web can only use one tree of resources. In order to be used for the pilot of the *HTCondor* batch system, the new system should support multiple resources.

e-groups CERN uses a system for nested user-groups called *e-groups*. It is possible for users to self-enrol in a group. The groups of users of LSF maps closely to e-groups. However, LSF Web doesn't natively support e-groups.

1.1.2 Cloudman

For my project I evaluated Cloudman as a tool to be used to configure the batch shares. Cloudman was developed at CERN as a very generic resource manager. It was designed to manage a complex cloud environment. It supports multiple tenants in a cloud with multiple data-centres, in multiple geographic locations.

During the evaluation we concluded that the data model used by Cloudman (appendix A.2) is too elaborate for our use cases. The system also has a number of bugs. We decided to start a new design incorporating the lessons learned from Cloudman.

1.2 Design considerations

We learned a lot from Cloudman and LSF Web. The most import choices made based on the earlier products are listed below.

Separation of concerns in Cloudman the security and validation logic was scattered throughout the code. Because of this it was very hard to verify that the code was working as intended. The new project should separate these concerns.

Tree structure in LSF Web the *allocations* are nested. In Cloudman the defined *resources* can have sub-resource types.

Both allocations and resources have a tree structure. Using a library for this tree structure makes some operations faster and easier to implement than when using adjacency lists.

Unit Tests there is non-trivial logic in the application when it comes to the structure of the allocation tree. The edge-cases should be tested with unit tests.

1.2.1 Django Implementation Choices

When setting up a project there are many small choices. We tried to follow the best practices from "two scoops of django"[1]. The other libraries were chosen by gut feeling.

Django Implementation Choices

cbv use class based views only when you can use them without customization *or* it enables code re-use.

tree library use the *treebeard*[2] library for trees.

object based permissions use the *django-permission*[3] library for object based permissions.

object factories use the *factory_boy*[4] library to create objects in tests.

coding standards the code should adhere to the *pep8* style guide as well as the checks of *flake8*[5]. Pre-commit hooks should be used to enforce this.

properties properties are stored as a JSON blob. This is a better fit than a key-value mapping.

[1] http://twoscoopspress.org/products/two-scoops-of-django-1-5
[2] https://tabo.pe/projects/django-treebeard/docs/2.0/
[3] http://django-permission.readthedocs.org/en/latest/
[4] http://factoryboy.readthedocs.org/en/latest/
[5] http://flake8.readthedocs.org/en/2.2.3/

2 User stories

2.1 Assign resources to a delegate

Subject allocation manager

The compute manager of the `theory` group needs to set up a share for the `theory_t3` group.

He logs into the system and creates a new allocation under his top level budget of capacity on `LSF shared`. The next morning the LSF configuration is updated and the users of `theory_t3` can use their new quota.

2.2 Easy enrolment for resource usage

Subject resource user

During a group meeting one of the summer students talks about his progress. He is disappointed that *LXPLUS* is too slow for his data processing tasks and he has trouble running the software from CERN at his home institution.

Someone tells him that he should use *LXBATCH* for his processing tasks. He gets access to the computing *e-group* of his section. His access is updated automatically.

2.3 User rights follow the allocation hierarchy

Subject computing manager

It is the middle of the summer. The quota of a sub-group needs to be updated but the group's manager is on vacation.

The manager of a group higher in the hierarchy updates the access rights for the sub-group.

2.4 Import from LSF Web

Subject Batch team

The TRAP project has been finished and the batch team decides to migrate from LSF Web to TRAP. There are major differences between LSF Web and TRAP:

- LSF Web contains user lists, not e-group names.
- TRAP supports multiple resource types, based on "compute" root types.

TRAP should provide users to migrate away from the old meta-data format (e.g. from user lists to e-group names).

It should be possible to import the data from LSF Web. This import should create a reasonable configuration for TRAP. This means that there is a tree of resource types from *compute* down to *lsf shares*.

The pledge of all groups is modelled in compute capacity. The group then assigns compute capacity to a *batch instance*. The top level shares within a batch instance are based on the raw compute capacity a group assigns to that batch instance.

2.5 Split a resource over subgroups

Subject computing manager

At some point in time, the majority of the computing capacity goes to two subgroups. The total amount of resources changes through time.

Since the total capacity needs to be "filled up", the children should be updated after new capacity is added. When capacity is "full", it can only be reduced after the children are reduced.

This means that it is almost impossible to reduce discrete capacity if it has been used. Users should only use discrete capacity if they are aware of the caveats.

In our opinion it is better to use proportional capacity values, but this does not map to some of the domains (i.e. virtual machine count).

3 MoSCoW

The system;

MUST:

support multiple projects the system should support multiple "projects" (e.g. *LSF shared*, *condor shared*).

check budget constraints the system should check and enforce the available budget when an allocation is updated or created.

support discrete allocation the allocation of a discrete resource (e.g. *CPU count*) should fit in its parents budget.

support proportional allocation proportionally assigned capacity is split over the descendants.

ensure the consistency of types the system should check that an allocation has a parent with a compatible resource type.

ensure consistent allocation modes the system should check that all siblings have the same "mode" of allocation (e.g. *discrete* or *proportional*).

contain the initial data from LSF Web the system should be able to import the share configuration from LSF Web. This is detailed in story 2.4.

SHOULD

be able to store properties it should be possible to store properties of an allocation. Ideally this storage should enable nested structures.

support allocation in the form of abstract computation capacity The system should enable users to split their *computing budget* over multiple projects.

support homogeneous trees the system should enable users to allocate, and should be capable of processing a configuration with siblings of different resource types.

provide an API the system should provide an API. This API should provide enough information for consumers to configure their system.

COULD

keep an audit log the system could keep an audit log with all the changes made to the allocations.

show status update the system could show the status + time of the last update of the configuration of a resource.

WON'T

support multi-zone cloud resources The system shall have no knowledge of multiple regions/zones for resources. Different (sub) locations are modelled as separate resources.

4 Design

We had multiple iterations of the database design for the *Tree Resource Allocation Project* (TRAP). In this section we explain the iterations of, and decisions behind the model.

The first model (figure 6) has a structure where allocations are nested. There is only one tree of allocations. The resource types and their conversion is implicit in the tree structure.

In the second iteration (figure 7) a *Cluster* has multiple resource types. Both *Clusters* and *Allocations* are nested. This provides the functionality from Cloudman. For example, the *OpenStack* cluster would have *Storage*, *CPU* and *Memory* resources.

In the third iteration (figure 8) the allocated amounts for each resource type are not stored in the JSON but in a separate many-to-many table. The constraints on the allocation of each resource of the Cluster need to be checked. This is a relatively complex feature.

We discussed the need for this use case and decided that for now, most resources could be normalized into classes with set properties (e.g. *medium EC2 compute instance*) instead of allocating all properties separately.

The fourth iteration includes this change (figure 9). Since multiple resources might use the same unit (e.g. *HS06*) this table was extracted.

4.1 Usage

The model mainly consists of *Resource* and *Allocation* objects. Both resources and allocations are nested.

Both resources and allocations are trees. This allows for a large degree of flexibility but it also makes the model harder to comprehend.

In figure 4 and 5 we show two modes of configuring the system. In both cases the root of the allocations consists of abstract computing power. In the left tree the split between HTCondor and LSF is made early on. In the right tree, this split is pushed down to the groups within experiments.

This sounds like an abstract choice. In practice this could be a useful feature when the capacity split between HTCondor and LSF can be updated automatically.

In the left figure *"The batch team chooses how our compute capacity is split between LSF and HTCondor"*. In the right figure *"User groups can choose how they split their compute capacity between LSF and HTCondor"*.

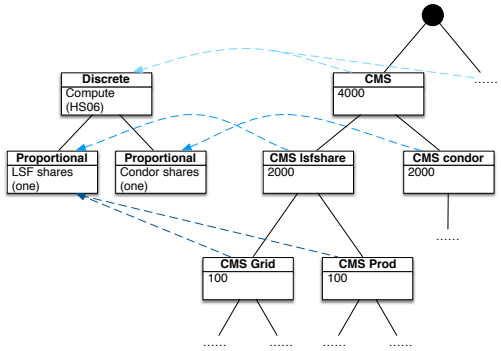

Figure 4: Allocation of *concrete* resource is pushed down

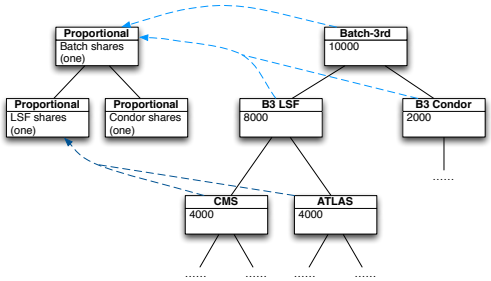

Figure 5: Resources are split at the top and then delegated.

4.1.1 Constraints

1. An allocation can only have the resource type of its parent *or* one of the child resource types of its parent.
2. If any allocation is of a discrete type, all its siblings should consist of a discrete resource type.
3. Siblings of a proportional allocation should be of the same resource-type.
4. A proportional resource-type can only have proportional descendants.

In the domain it *does* make sense to have a discrete type with a proportional assignment below it. This could be used to, for example, give half the virtual machines to one group and the rest to others, without the need to update this split after the top amount changes.

However since reducing a discrete allocation is hard (read story 2.5), we restrict this possibility. The restriction follows from a constraint that a proportional type can only have proportional descendants.

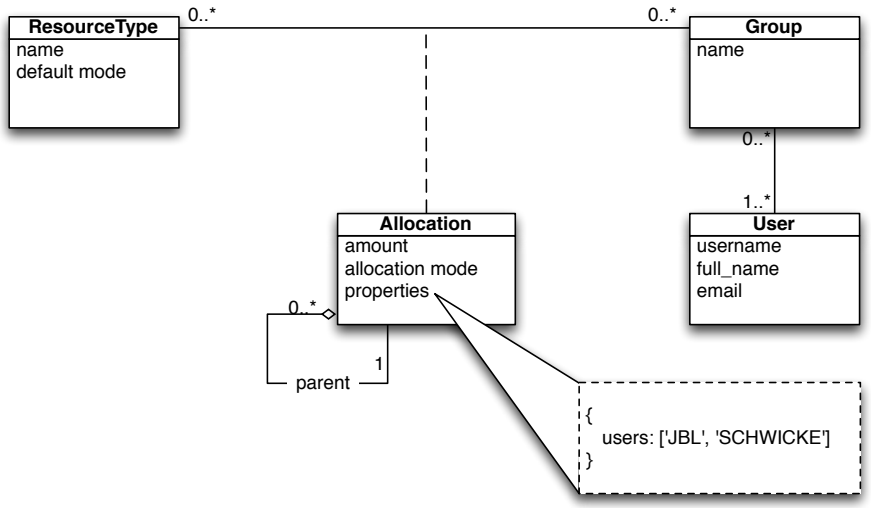

Figure 6: First iteration of TRAP model

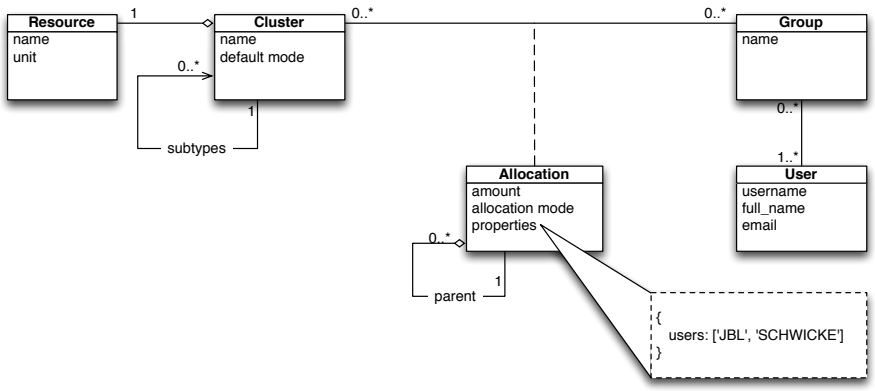

Figure 7: Minimal viable model — Cloudman

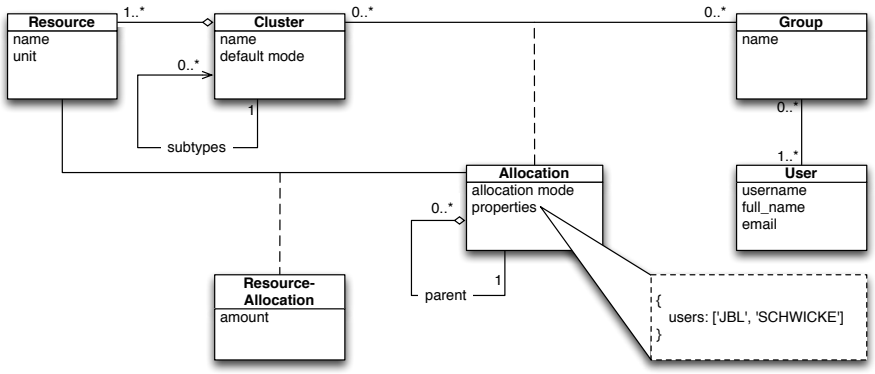

Figure 8: Minimal viable model — Multiple resource types in cluster

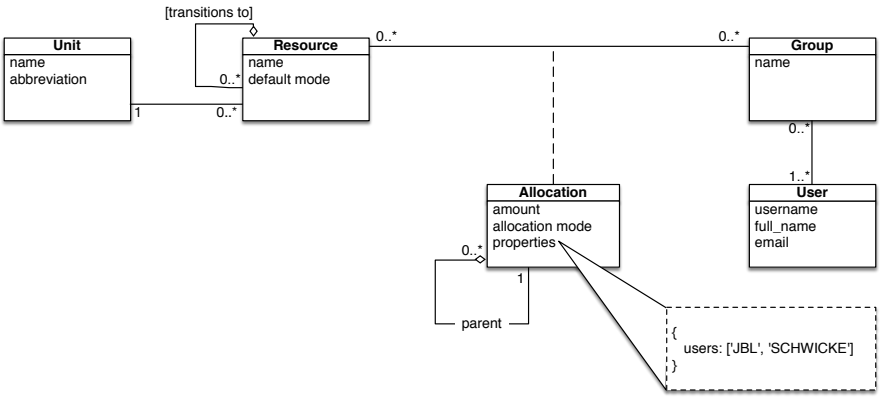

Figure 9: Minimal viable model — One resourcetype per allocation

5 Dependencies

Most of the python dependencies are listed in the requirements files (`requirements/base.txt`). Some of the dependencies have not been released recently and a git version was needed in orer to use them. Those are listed below:

```
pip install git+https://github.com/Brown-University-Library/\
        django-shibboleth-remoteuser.git
```

6 Security

The security and authorization system is based on the *django-permission* library (see section 1.2.1). As mentioned in story 2.3 the user rights should follow the structure of the tree.

An allocation can be changed by one of its administrators or the administrator of one of its ancestors. For reference, listing 1 shows the usage of this rights system.

```
from allocation.models import *
from django.contrib.auth.models import User
batch = Allocation.objects.first()
user = User.objects.get(username='username')

# clear all groups, add him to top atlas group
user.groups.clear()
user.groups.add(Group.objects.filter(name='atlas'))
user.save()

# The user has no top level batch permissions
user.has_perm('allocation.add_allocation', batch)
# => False

# But the user has permissions on one of the child
# groups in atlas.
atlas_tzero = Group.objects.get(name='u_ATLASTZERO2')
user.has_perm('allocation.add_allocation', atlas_tzero)
# => True

# In a view, we use decorators to check access. The
# decorator implicitly takes the id from the instance
# variables
@permission_required('allocation.change_allocation',
                     raise_exception=True)
class AllocationUpdate(views.FormValidMessageMixin,
                       UpdateView):
    form_valid_message = "Allocation has been updated."

    model = Allocation
    form_class = AllocationAmountDescriptionAdminForm
```

Listing 1: Sample code that shows the recursive permissions.

7 Conclusions

At the moment, TRAP is a proof of concept application that lacks some of the features that were wished for. After choosing to re-implement the product there was not enough time left to deliver all the features.

The application still needs to be deployed and integrated into the configuration of LSF.

Its design is (relatively) clean and extensible, and the allocation logic is covered with unit tests. It should be relatively simple (compared to Cloudman or LSF Web) to implement and migrate to this new project.

A Database schemas

A.1 LSF Web database schema

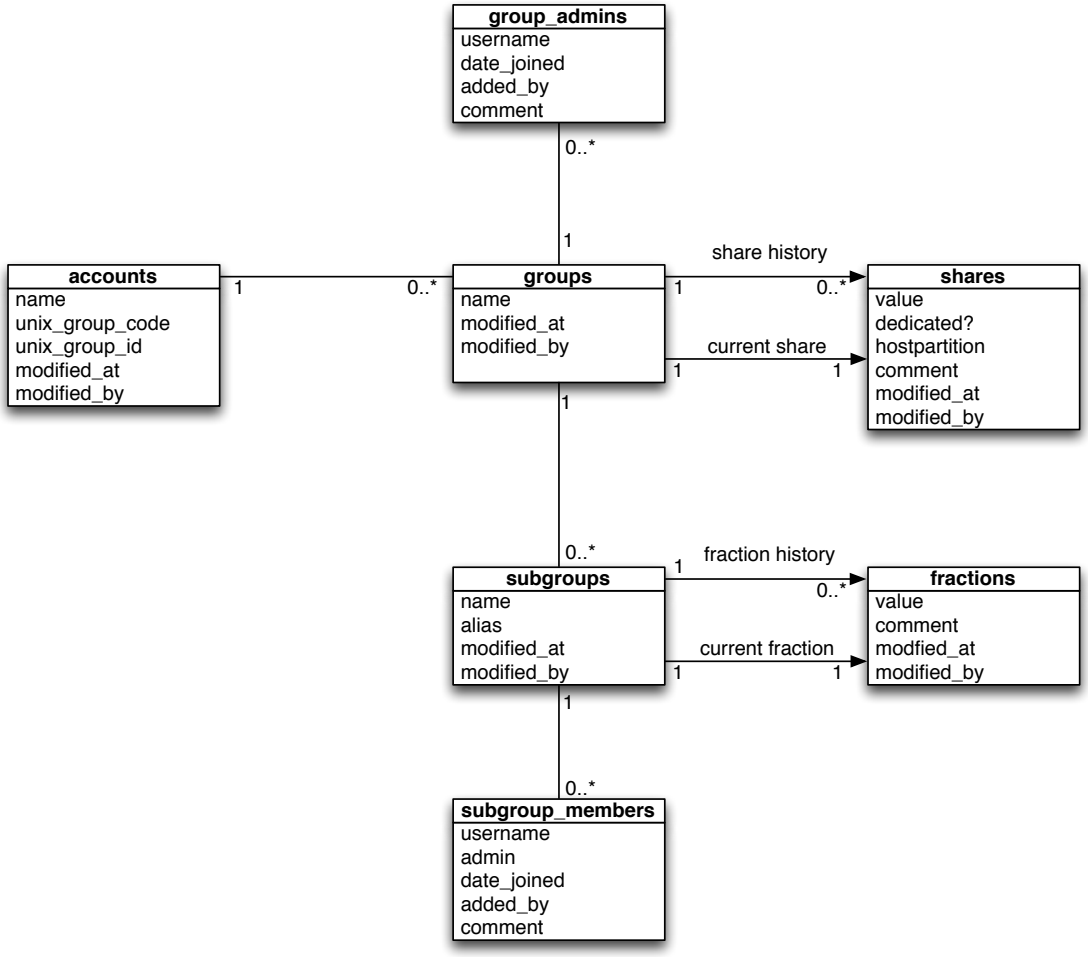

Figure 10: Database schema from LSF Web

A.2 Cloudman database schema

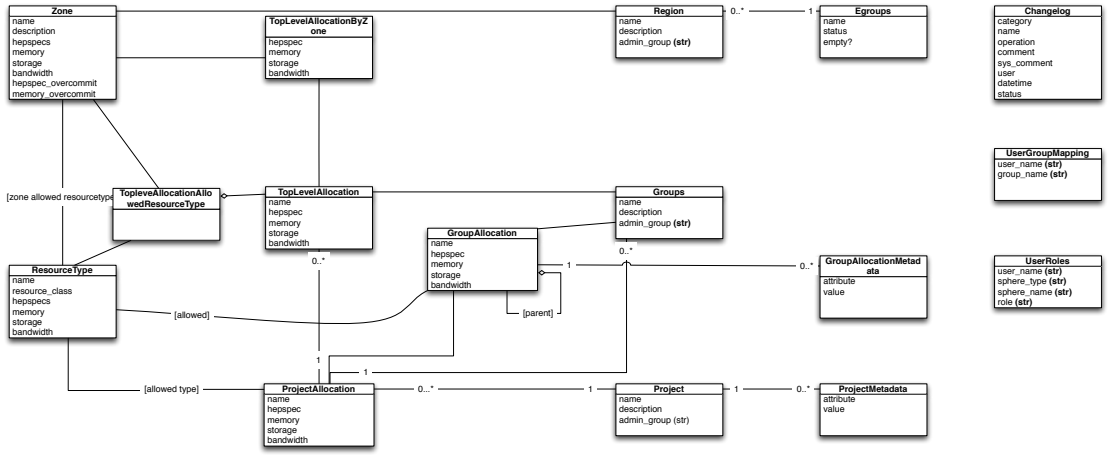

Figure 11: Database schema from Cloudman