Perspective for nova scheduler.

Alexey Ovchinnikov, Boris Pavlovic.
Mirantis, Inc. 2013.

Updated on 13.08.2013.

Intro

One of hot topics in nova discussion lately is a multitude of proposals for scheduler awareness of various compute node and cloud characteristics. Those are network awareness, utilisation awareness and extended node state awareness to name a few. All such mechanisms that allow finer cloud tuning are good for improving both flexibility and performance and definitely should be considered. Unfortunately limitations of present implementation of nova scheduler may be a major roadblock on the way of named features.

In this proposal we aim to discuss

a. current implementation of nova scheduler,
b. point out its limitations and
c. outline a way to circumvent current problems.

Let us start with a quick overview of what we have now.

1. Nova scheduler overview

Now there are two possible bottlenecks in current implementation of nova scheduler. The first one is host state storage, the second one is host state update. Let us consider how this two tasks can affect scheduler performance in more detail. To do this let us consider first how scheduler acquires data needed for fulfilling its task namely to find a host that fits the instance to be run.

To be able to make a desision scheduler needs to take into account resources available on all hosts present in the cloud. Obviously it needs to get this resources data from somwhere first. Thus when scheduling an instance nova scheduler issues a request to database to retrieve resources available on all compute nodes. This db request has 2 joins:

```
def compute_node_get_all(context):
      return model_query(context, models.ComputeNode).\
      options(joinedload('service')).\
      options(joinedload('stats')).\
      all()
```

After host states are obtained scheduler iterates over all of them several times to filter and weight hosts list. Now a relatively small amount of resources is tracked, namely RAM, CPU and disk usage and some others. Each time a user wants to create an instance all data concerning compute nodes gets fetched from database by nova scheduler via *compute_node_get_all()* call. Even with such a small amount of resources to be tracked as is tracked now scheduling may become a long and expensive task.

As it was mentioned earlier node resources are stored in a database, and it is compute node's responsibility to update its state on a regular basis. By default it sends state update to DB once in a minute and each time an instance is created on this node. As more nodes are added to the cloud more DB updates take place simultaneously and more data has to be exchanged with DB. And this in turn slows scheduling down and keeps an instance in 'Scheduling' state i.e. in state in which it can not perform any meaningful actions longer.

To be more specific we conducted a series of experiments which model interactions with database during normal cloud operation. Database was populated with mock entries for compute nodes and corresponding compute node states. Each node had ten corresponding states to reflect upcoming increase of compute node characteristics to be tracked as to extend flexibility one has to add extra parameters to host state which are tracked with *compute_node_stats* table. Then each entry was updated over network roughly once in a minute to emulate compute node state updates. Average time needed to perform single *compute_node_get_all()* call was measured. This time essentially determines duration of a single boot request. The experiment was carried out for number of nodes varying from 500 to 10000. Graphic representation of obtained results can be found in fig.1.

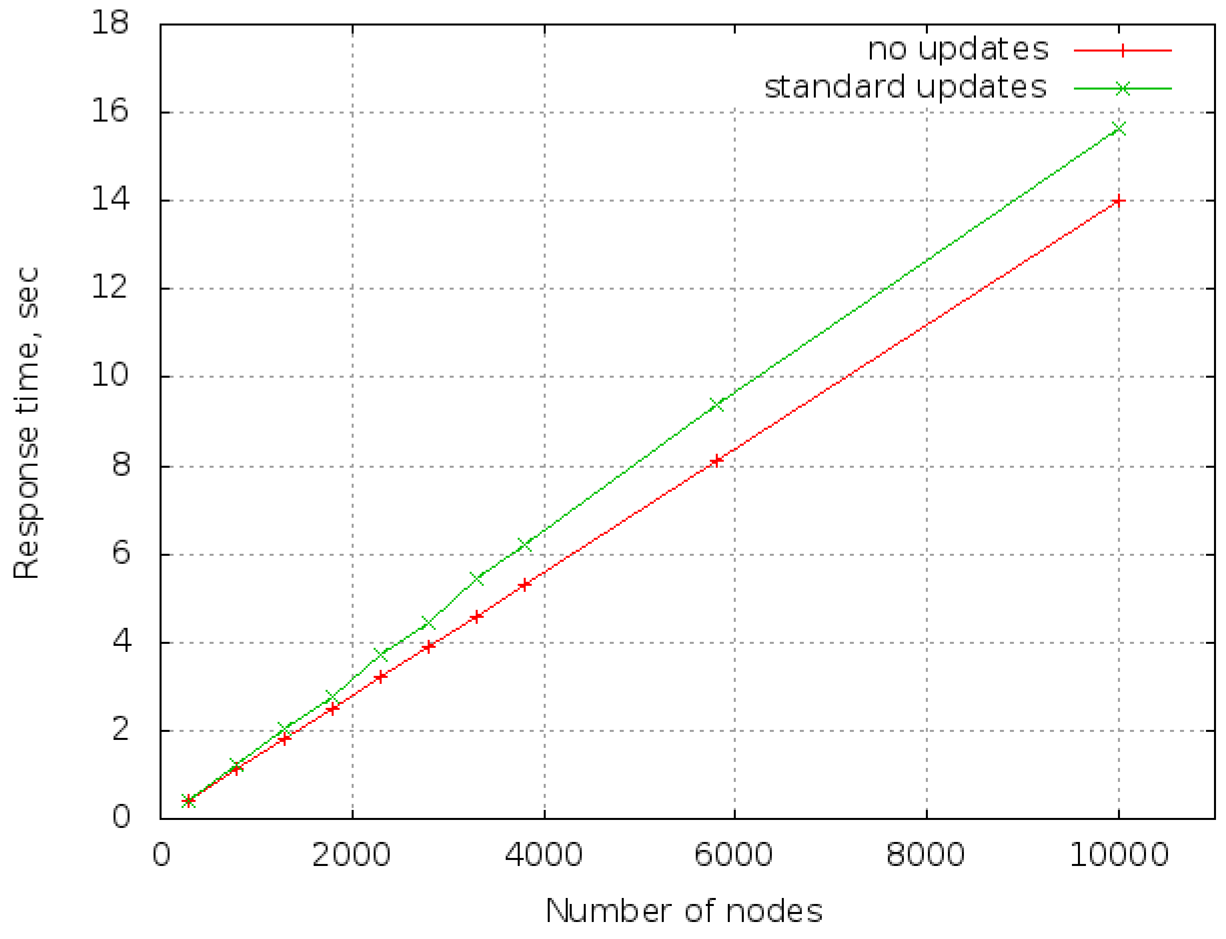

Fig.1. Averaged for 25 calls *compute_node_get_all()* call time for static tables (red line) and tables that get regular updates (green line).

While linearity of response time plots is trully remarkable and call time does not increase significantly over base case even when the load on mysql is big, time required for single boot is rather long even in relatively small experimental sets. It must be remembered that this boot requests can't be processed in parallel, i.e. if a user issue boot command while another one is still not over he has to wait two times longer in the worst case. In case when several users interact with a cloud simultaneously cumulative latency may become very big even for thousands of nodes. It is also worth to remember that in our experiment only fairly simple type of interactions has been covered. It still may (and eventually will) happen that a single boot request issues several DB access requests, for example to cinder or glance databases. Another issue that is not yet covered, but has a great potential of slowing scheduling down is filtering and weighting in case of multi-instance boot request. Adding extra schedulers to the cloud won't solve the problem as this will increase DB load on one hand (due to the fact that JOIN requests don't scale very well) and introduce extra possibilities for racing.

2. We are sufficiently scared now, but what can we do?

Let us go over scheduler interaction with host states once again. First compute node publishes its state in DB via RPC call, then on a boot request scheduler picks all data on compute node states from database and creates a local objects representing node states. This scheme looks redundant even if we haven't known how bad DB access time can be. Thus we propose to eliminate DB access and publish compute node state directly to schedulers queue. This will actually reduce network load as now we need to transfer data only once -- to scheduler not copying it to DB. Also it should be noted that in our approach scheduler does not need to waste time populating the pool of host states representing objects in memory on every boot request because it already has them there.

Summing things up we propose

- to eliminate host state database
- to make compute nodes inform scheduler directly
- to make scheduler responsible for storing up to date compute nodes state.

While this proposals may look rather radical, the real question behind them is

3. How can we do it?

Scheduler should store and update data about host states. As data is temporary we could use in-memory storage like memcached and store in it key:values pairs where keys are host names and values are json object representing host state. This may look like a surprise, but as for the moment almost everything that is needed already exists in nova scheduler. To implement this proposal we have to make rather smallish changes in scheduler and compute manager.

In scheduler:

- Add storage for host state data in memcached. (Add an accessor class with three methods: get(key), set(key, value) and get_all()).
- Add scheduler RPC method for updating host state in memcached. It is important to note that this is not going to be a fanout method, but a call so each host state update get processed only once while several schedulers could listen to this queue and process messages in parallel.
- Change *nova.scheudler.host_manager.py:get_all_host_state()* to use key:value storage for host resources (e.g. memcached). Specifically this means that we will use in-memory cache instead of database for storing host resources which can be done with replacing one line in *nova.scheudler.host_manager.py:get_all_host_state()* (compute_nodes = db.compute_node_get_all(context) → compute_nodes = self.host_cache.get_all(self.node_names)).

In compute manager:

- scheduler.compute_node_update() should be called instead of db.compute_node_update()

4. What we get after this change?

Despite being minimal this changes can bring sufficient improvements. At least some of them are:

- **Reduce DB load.** As we don't need to go to DB each time we want to boot an instance and more important we don't go to DB each time we want to update host state we produce less DB queries.
- **Reduce traffic.** As we need no more to toss host states back and forth overall network load becomes smaller.
- **Number of RPC calls stays the same.** Despite that our proposed solution relies on RPC calls, their number does not change as all we do is RPC call optimization. Now compute node makes RPC call to conductor to access DB, in proposed approach RPC call goes directly to scheduler.
- **Greater flexibility.** This approach will allow to track more host state parameters much easier and to pick hosts with greater precision.
- **Scalability.** This approach requires relatively small resources as 10k host states can fit in a few tens of megabytes.
- **Scheduler speedup.** DB access induced latency goes away with DB access thus allowing to schedule instances faster.

While all points mentioned above are interesting great flexibility deserves more discussion.

5. What does Greater flexibility mean?

5.1. Use different source of data for scheduling

Right now a user is somewhat constrained in his choice of nodes on which to run an instance. While it is possible to make a filter to affect node choice, it can't be done efficiently in many cases. Consider for example volume affinity. Sometimes OpenStack users want to associate instances with specific volumes to improve performance of their VMs. To achieve this now one can make a custom filter which has to access cinder db to get info about specified volumes. When everything host-specific is stored in one place this extra DB access gets eliminated. Similar interest has been seen recently in ability to filter hosts according to their physical location i.e. association with specific server rack. This can again be possibly done now with a clever filter accessing neutron DB, but in the framework of proposed approach no extra DB call will be needed.

5.2. Much simpler and efficiently work around compute node resources

PCI passthrough can be considered as a characteristic example of a new feature screaming for scheduler optimisation.

To implement PCI passthrough i.e. to add ability to directly connect PCI device to an instance scheduler must know all about all PCI devices on each node and by all we mean addresses, aliases, statuses, vendor ids and many other parameters. Now we have two options to store this device-specific data:

- **Store all devices in compute_node_stats as json**
  This will produce tons of race conditions and will require tons of dirty hacks. Also we can face another problem, namely “values” field length is limited. So we can end up not being able to store all devices data. And if we decide to store each device in a separated extra spec then we run straight into scalability problem.

- **Store all devices in separated table and join it to compute_node table**
  In this case code remains clean, but we face serious performance issues because there are use cases where we have up to 1k devices on each compute node (consider for example NICs with support of SR-IOV which means that devices have up to 128 VF each of which must be considered as a separate device). So apparently this won’t work even for a cloud with just a thousand nodes.

Our approach solves both problems:

- we don’t store anything in DB so we don’t have problems with scalability
- it is extremely easy to add arbitrary extra state to a json object used to pass host state to scheduler.

In conclusion we can gain performance boost by just reorganizing the way compute nodes notify scheduler about their state. Our approach reduces DB load and network traffic as well as adds a convenient way to pass extra node characteristics to scheduler. It is important to stress once more that our approach does not inflict any extra costs neither in memory, nor in number of RPC calls.