Dynamic Priority Scheduler

Table of contents

1 Purpose..... 2
2 Features.....2
3 Problem Addressed..... 2
4 Approach..... 3
5 Comparison to other Hadoop Schedulers..... 3
6 Implementation and Use..... 4
6.1 Accounting..... 4
6.2 Preemption..... 4
6.3 Security..... 4
6.4 Control..... 5
6.5 Configuration..... 6
6.6 Examples..... 7
7 Bibliography..... 9

1 Purpose

This document describes the Dynamic Priority Scheduler, a pluggable MapReduce scheduler for Hadoop which provides a way to automatically manage user job QoS based on demand.

2 Features

The Dynamic Priority Scheduler supports the following features:

- Users may change their dedicated queue capacity at any point in time by specifying how much they are willing to pay for running on a (map or reduce) slot over an allocation interval (called the spending rate).
- Users are given a budget when signing up to use the Hadoop cluster. This budget may then be used to run jobs requiring different QoS over time. QoS allocation intervals in the order of a few seconds are supported allowing the cluster capacity allocations to follow the demand closely.
- The slot capacity share given to a user is proportional to the user's spending rate and inversely proportional to the spending rates of all the other users in the same allocation interval.
- Preemption is supported but may be disabled.
- Work conservation is supported. If a user doesn't use up the slots guaranteed based on the spending rates, other users may use these slots. Whoever uses the slots pays for them but nobody pays more than the spending rate they bid.
- When you don't run any jobs you don't pay anything out of your budget.
- When preempting tasks, the tasks that have been running for the shortest period of time will be killed first.
- A secure REST XML RPC interface allows programmatic access to queue management both for users and administrators. The same access control mechanism has also been integrated in the standard Hadoop job client for submitting jobs to personal queues.
- A simple file-based budget management back-end is provided but database backends may be plugged in as well.
- The scheduler is very lightweight in terms of queue memory footprint and overhead, so a large number of concurrently serving queues (100+) are supported.

3 Problem Addressed

Granting users (here MapReduce job clients) capacity shares on computational clusters is typically done manually by the system administrator of the Hadoop installation. If users want more resources they need to renegotiate their share either with the administrator or the competing users. This form of social scheduling works fine in small teams in trusted environments, but breaks down in large-scale, multi-tenancy clusters with users from multiple organizations.

Even if the users are cooperative it is too complex to renegotiate shares manually. If users' individual jobs vary in importance (criticality to meet certain deadlines) over time and between job types severe allocation inefficiencies may occur. For example, a user with a high allocated capacity may run large low-priority test jobs starving out more important jobs from other users.

4 Approach

To solve this problem we introduce the concept of dynamic regulated priorities. As before each user is granted a certain initial quota, which we call budget. However, instead of mapping the budget to a capacity share statically we allow users to specify how much of their budget they are willing to spend on each job at any given point in time. We call this amount the spending rate, and it is defined in units of the budget that a user is willing to spend per task per allocation interval (typically < 1 minute but configurable).

The share allocated to a specific user is calculated as her spending rate over the spending rates of all users. The scheduler is preempting tasks to guarantee the shares, but it is at the same time work conserving (lets users exceed their share if no other tasks are running). Furthermore, the users are only charged for the fraction of the shares they actually use, so if they don't run any jobs their spending goes to 0. In this text a user is defined as the owner of a queue which is the unit of Authentication, Authorization, and Accounting (AAA). If there is no need to separate actual users' AAA, then they could share a queue, but that should be seen equivalent to users sharing a password, i which in general is frowned upon.

5 Comparison to other Hadoop Schedulers

Hadoop-on-demand (HOD) allows individual users to create private MapReduce clusters on demand to meet their changing demands over time. Prioritization across users is still a static server configuration, which in essence still relies on users being cooperative during high-demand periods. Further, HOD breaks the data-local scheduling principle of MapReduce and makes it more difficult to efficiently share large data sets.

The fair-share (FAIR), and capacity (CAP) schedulers are both extensions of a simple fifo queue that allow relative priorities to be set on queues. There is no notion of charging or accounting on a per-use basis, and shares cannot be set by the users (they have to be negotiated a-priori). So although sharing data is more efficient than with HOD, these schedulers suffer from the same kind of social scheduling inefficiencies.

Amazon Web Services (AWS) Elastic MapReduce allows virtual Hadoop clusters, which read input from and writes output to S3, to be set up automatically. New instances can be added to or removed from the cluster to scale the jobs up or down. Our approach is more lightweight in that the virtualization is not on an OS level but on a queue level. This allows us to react faster to temporary application bursts and to co-host more concurrent users

on each physical node. Furthermore, our approach implements a demand-based pricing model whereas AWS uses fixed pricing. Furthermore, our solution works in any Hadoop cluster, not only in an AWS environment. Finally, with AWS you can get up and running and reconfigure within the order of 10 minutes. With this scheduler you can get started and reconfigure within seconds (specified with alloc-interval below).

6 Implementation and Use

6.1 Accounting

Users specify a spending rate. This is the rate deducted for each active/used task slot per allocation interval. The shares allocated are not calculated directly based on the user specified spending rate but the effective rate that is currently being paid by users. If a user has no pending or running jobs the effective spending rate for that user is set to 0, which assures that the user is not charged anything against her budget. If a job is pending the effective rate is set to the user-specified rate to allow a share to be allocated so the job can be moved from pending to active state.

The number of map tasks and reduce tasks granted to a user in each allocation interval is based on the effective spending rate in the previous allocation interval. If the user only uses a subset of the allocated tasks only that subset is being charged for. Conversely, if the user is able to run more tasks than the granted quota due to other tasks not running up to their full quota only the spending rate times the quota is being charged for. The price signaled to users so they can estimate shares accurately is based on the effective rates.

6.2 Preemption

Tasks that are run in excess of the quota allocated to a particular user are subject to preemption. No preemption will occur unless there are pending tasks for a user who has not fulfilled her full quota. Queues (users) to preempt are picked based on the latest start- time of jobs in excess of their quota.

Excess jobs in a queue will not be killed unless all excess jobs have been killed from queues with later start times. Within a queue that is exceeding its quota the tasks that have run the shortest time will be killed first. All killed tasks are put back in a pending state and will thus be resubmitted as soon as existing tasks finish or the effective share for the queue goes up, e.g. because other users' jobs finish. The preemption interval may be set equal to the scheduling interval, a longer interval, or 0 in which case no preemption (task killing) is done.

6.3 Security

If any user can impersonate any other user or if any user can submit to any queue, the economic incentives of the mechanism in the scheduler breaks down and the problem of

social scheduling and free-riding users comes back. Hence, stronger authentication and authorization is required in this scheduler compared to the other schedulers. Since the authz mechanisms are still being worked out in Hadoop we implemented a simple shared secret signature based protocol inspired by the AWS query (REST) API authentication used for EC2 services.

Simple guards against replay attacks and clock/nonce synchronization are also available but the main idea is to require that users prove to have a secret key to be allowed to submit jobs or control queue spending rates. There are currently two roles implemented, users and administrators. In general users have access to information such as the current price (aggregate spending rates of active queues) of the cluster and their own queue settings and usage, and can change the spending rate of their queue. Admin users may also add budgets to queues, create and remove queues, and get detailed usage info from all queues.

The basic authentication model relies on the clients calculating a HMAC/SHA1 signature across their request input as well as a timestamp and their user name, using their secret key. The server looks up the user's secret key in an acl and determines the role granted after verifying that the signature is ok. For the REST API discussed next the signature is passed in the standard HTTP Authorization header and for the job submission protocol it is carried in a job configuration parameter (mapred.job.signature).

If stronger authentication protocols (asynchronous keys) are developed for Hadoop at least the job submission protocol will be adopted to use it.

6.4 Control

The job tracker installs a servlet accepting signed HTTP GETs and returning well-formed XML responses. The servlet is installed in the scheduler context (like the fair-share scheduler UI)

It is installed at [job tracker URL]/scheduler

HTTP Option	Description	Authz Required
price	Gets current price	None
time	Gets current time	None
info=`queue`	Gets queue usage info	User
info	Gets queue usage info	User
infos	Gets usage info for all queues	Admin
setSpending=`spending` &queue=`queue`	Set the spending rate for queue	User

HTTP Option	Description	Authz Required
addBudget=`budget` &queue=`queue`	Add budget to queue	Admin
addQueue=`queue`	Add queue	Admin
removeQueue=`queue`	Remove queue	Admin

Example response for authorized requests:

```
<QueueInfo>
  <host>myhost</host>
  <queue name="queue1">
    <budget>99972.0</budget>
    <spending>0.11</spending>
    <share>0.008979593</share>
    <used>1</used>
    <pending>43</pending>
  </queue>
</QueueInfo>
```

Example response for time request:

```
<QueueInfo>
  <host>myhost</host>
  <start>1238600160788</start>
  <time>1238605061519</time>
</QueueInfo>
```

Example response for price request:

```
<QueueInfo>
  <host>myhost</host>
  <price>12.249998</price>
</QueueInfo>
```

Failed Authentications/Authorizations will return HTTP error code 500, ACCESS DENIED: `query string`

6.5 Configuration

Option	Default	Comment
mapred.jobtracker. taskScheduler	Hadoop FIFO scheduler	Needs to be set to org.apache.hadoop.mapred. DynamicPriorityScheduler

Option	Default	Comment
mapred.priority-scheduler. kill-interval	0 (don't preempt)	Interval between preemption/kill attempts in seconds
mapred.dynamic-scheduler. alloc-interval	20	Interval between allocation and accounting updates in seconds
mapred.dynamic-scheduler. budget-file	/etc/hadoop.budget	File used to store budget info. Jobtracker user needs write access to file which must exist.
mapred.priority-scheduler. acl-file	/etc/hadoop.acl	File where user keys and roles are stored. Jobtracker user needs read access to file which must exist.

Budget File Format (do not edit manually if scheduler is running, then servlet API should be used):

```
user_1 budget_1 spending_1
...
user_n budget_n spending_n
```

ACL File Format (can be updated without restarting Jobtracker):

```
user_1 role_1 key_1
...
user_n role_n key_n
```

`role` can be either admin or user.

6.6 Examples

Example Request to set the spending rate

```
http://myhost:50030/scheduler?setSpending=0.01&queue=myqueue
```

The Authorization header is used for signing

The signature is created akin to the AWS Query Authentication scheme

```
HMAC_SHA1("[query path]&user=[user]&timestamp=[timestamp]", key)
```

For the servlet operations query path is everything that comes after /scheduler? in the url. For job submission the query path is just the empty string "".

Job submissions also need to set the following job properties:

```
```

```
   -Dmapred.job.timestamp=[ms epoch time]
   -Dmapred.job.signature=[signature as above]
   -Dmapreduce.job.queue.name=[queue]
```

Note queue must match the user submitting the job.

Example python query

```
import base64
import hmac
import sha
import httplib, urllib
import sys
import time
from popen2 import popen3
import os

def hmac_sha1(data, key):
    return urllib.quote(base64.encodestring(hmac.new(key, data, sha).digest()).strip())

stdout, stdin, stderr = popen3("id -un")
USER = stdout.read().strip()
f = open(os.path.expanduser("~/.ssh/hadoop_key"))
KEY = f.read().strip()
f.close()
f = open(os.path.expanduser("/etc/hadoop_server"))
SERVER = f.read().strip()
f.close()
URL = "/scheduler"
conn = httplib.HTTPConnection(SERVER)
params = sys.argv[1]
params = params + "&user=%s&timestamp=%d" % (USER,long(time.time()*1000))
print params
headers = {"Authorization": hmac_sha1(params, KEY)}
print headers
conn.request("GET",URL + "?" + params,None, headers)
response = conn.getresponse()
print response.status, response.reason
data =  response.read()
conn.close()
print data
```

Example python job submission parameter generation

```
import base64
import hmac
import sha
import httplib, urllib
import sys
import time
import os
from popen2 import popen3

def hmac_sha1(data, key):
    return urllib.quote(base64.encodestring(hmac.new(key, data, sha).digest()).strip())
```

```
stdout, stdin, stderr = popen3("id -un")
USER = stdout.read().strip()
f = open(os.path.expanduser("~/.ssh/hadoop_key"))
KEY = f.read().strip()
f.close()
if len(sys.argv) > 1:
  params = sys.argv[1]
else:
  params = ""
timestamp = long(time.time()*1000)
params = params + "&user=%s&timestamp=%d" % (USER,timestamp)
print "-Dmapred.job.timestamp=%d -Dmapred.job.signature=%s -Dmapreduce.job.queue.name=%s" %
 (timestamp, hmac_sha1(params, KEY), USER)
```

7 Bibliography

T. Sandholm and K. Lai. **Mapreduce optimization using regulated dynamic prioritization**. In SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, pages 299-310, New York, NY, USA, 2009.

T. Sandholm, K. Lai. **Dynamic Proportional Share Scheduling in Hadoop**. In JSSPP '10: Proceedings of the 15th Workshop on Job Scheduling Strategies for Parallel Processing, Atlanta, April 2010.

A. Lenk, J. Nimis, T. Sandholm, S. Tai. **An Open Framework to Support the Development of Commercial Cloud Offerings based on Pre-Existing Applications**. In CCV '10: Proceedings of the International Conference on Cloud Computing and Virtualization, Singapore, May 2010.