This is based on Ionel's thesis, specifically related to support for complex constraints within Firmament. In the thesis Ionel mentions that complex constraints such as Pod-to-Pod Affinity/Anti-Affinity create mutual dependencies between several tasks and machines. Min-cost flow schedulers consider entire workloads in each scheduling round, albeit they do not support complex constraints. The flow networks these schedulers generate do not encode dependencies between tasks and machines, and thus solvers route tasks flow supply to sink nodes independently.

Currently, there is no min-cost flow cluster scheduler that offers complex constraints. Prior work claims that complex constraints cannot be expressed in flow networks. Only way possible, as suggested in Malte's thesis, to address pod-to-pod affinity complex constraints is by processing one pod at a time using multi-scheduling rounds.

However, in Ionel's thesis, he demonstrated that this is not the case: complex constraints can be represented in flow networks using new flow constructs as introduced in the Appendix section below, as a reference.

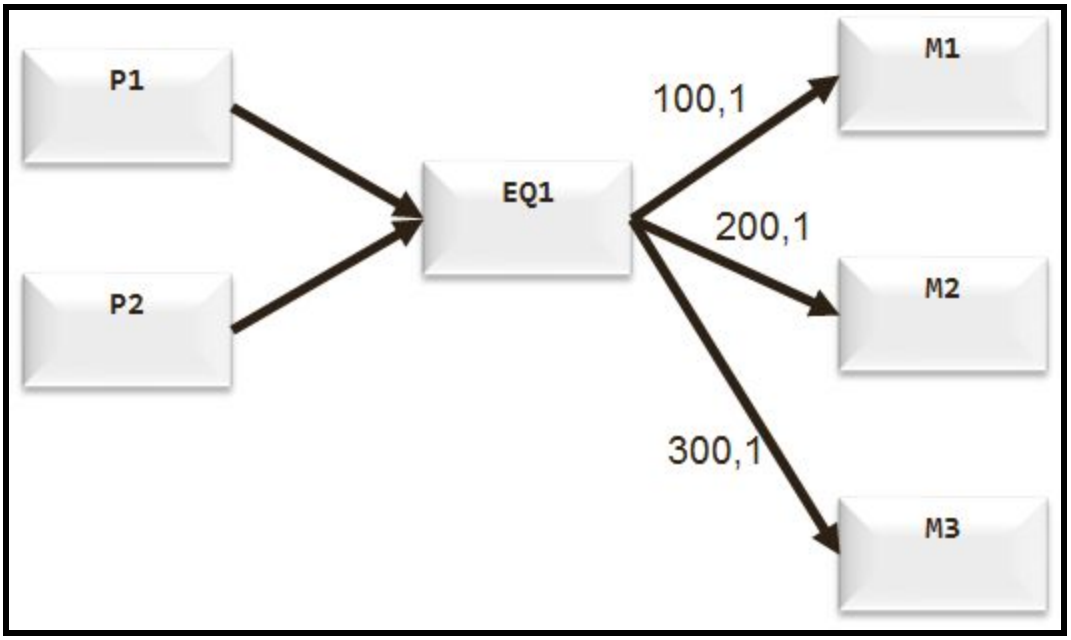
This document is to capture questions/issues as we are trying to understand how it all works.

Anti-Affinity Scenario-1: Let us consider an anti-affinity scenario for 2 tasks using the "xor" flow network construct. In this scenario:

- Pod P1 has anti-affinity with P2 (anti-affinity label value is: P2)
- Pod P2 has anti-affinity with P1 (anti-affinity label value is: P1)

These Pods can be grouped together by using hash for Pod P1 as (hash-function)(P1 + P2). Here P2 is the value of anti-affinity label within Pod P1 definition. Similarly, hash for Pod P2 as (hash-function)(P2 + P1).

As both hash values are going to have same value, they can be grouped together. Based on the "xor" construct approach as shown in the following diagram, Pods P1 & P2 will be placed on separate machines in order to meet the anti-affinity requirements.



Issue: Although, in case there are multiple Pods P1 & P2 coming in during the same scheduling round, only one instance of these P1 & P2 pods would be scheduled. Rest will remain unscheduled.

Anti-Affinity Scenario-2: Let us consider an anti-affinity scenario for 3 tasks using the "xor" flow network construct. In this scenario:

- Pod P1 has anti-affinity with P2 (anti-affinity label value is: P2)
- Pod P2 has anti-affinity with P1 (anti-affinity label value is: P1)
- Pod P2 has anti-affinity with P3 (anti-affinity label value is: P3)
- Pod P3 has anti-affinity with P2 (anti-affinity label value is: P2)

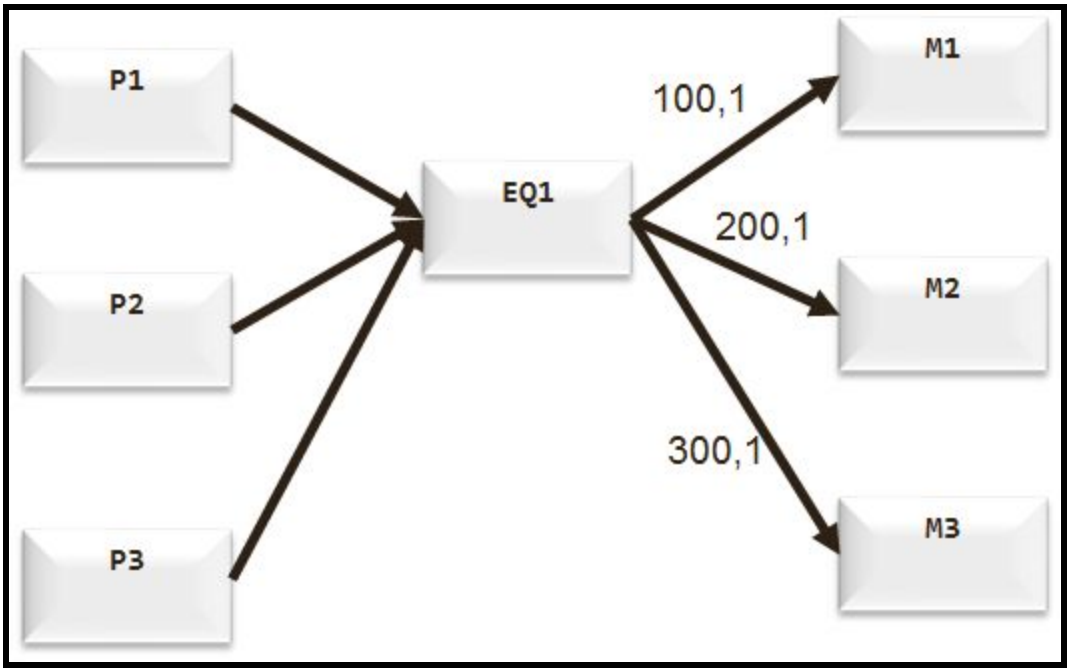
Issue: Unlike the previous scenario, it is not clear how to group Pods in this case. Based on our earlier hash-function approach, we will end up having two distinct groups of Pods (P1 & P2) and (P2 & P3). In this case, Pod P2 shows up in two different groups, which is not a valid flow network. In order for flow network to work, all these three Pods need to be grouped together as shown in the graph below. It is not clear how to accomplish this, at this time.

In any case, let us assume that we are able to group these three Pods together somehow as shown in the graph below. Based on the "xor" construct approach as shown in the following diagram, Pods P1, P2 & P3 will be placed on separate machines in order to meet the anti-affinity requirements.

Issue: It is important to highlight that Pod 3 is also forced to run on a different machine, even though it can run on machine where Pod 1 could run as it does not have anti-affinity with Pod 1.

Issue: It is also possible that due to different resource requirements for these three Pods, only machines M1 & M2 can accommodate these Pods. As the maximum capacity on the arcs is of value 1, one of the Pods will remain unscheduled as there are two machines available and there are 3 incoming Pods.

Issue: Finally, as the number of pods having anti-affinity with each other increases, grouping of Pods becomes near to impossible.



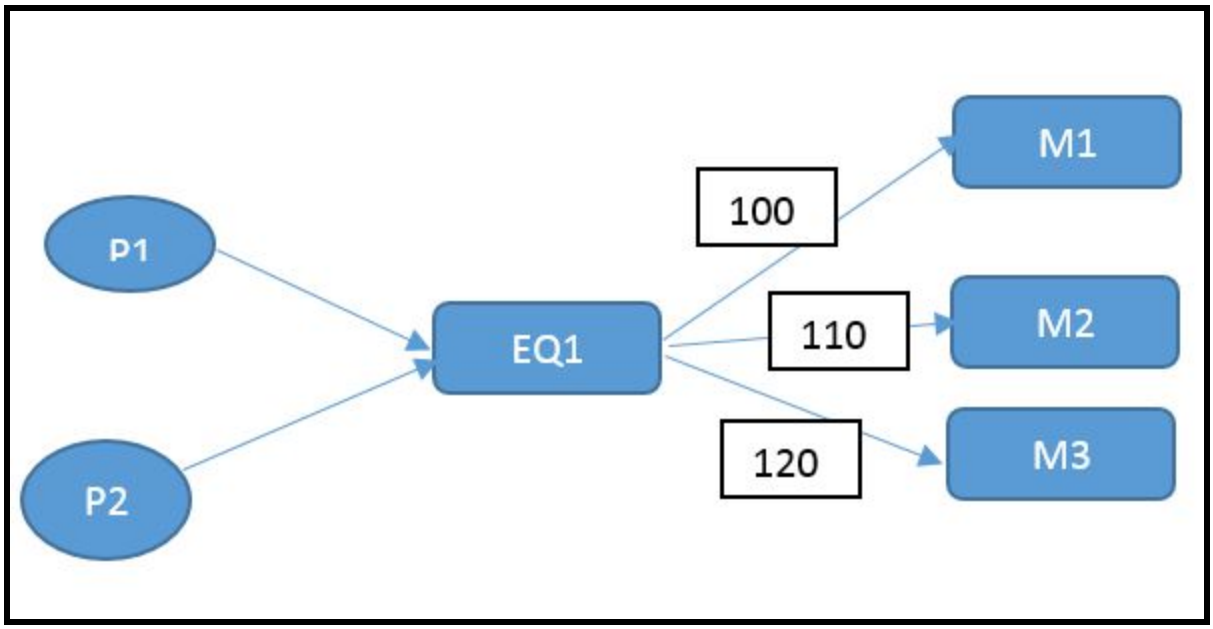
Affinity Scenario: Let us consider an affinity scenario for 2 tasks. In this scenario:

- Pod P1 has affinity with P2 (affinity label value is: P2)
- Pod P2 has affinity with P1 (affinity label value is: P1)

These two Pods can be easily grouped together by using hash for Pod P1 as (hash-function)(P1 + P2). Here P2 is the value of affinity label within Pod P1 definition. Similarly, hash for Pod P2 as (hash-function)(P2 + P1).

As both hash values are going to have same value, they can be grouped together. Based on the following diagram, Pods P1 & P2 will be placed on a same machine in order to meet the affinity requirements.

Issue: Although, in case there are multiple Pods P1 & P2 coming in during the same scheduling round, all of them will end up getting placed on machine M1 as it has the lowest cost on the arc. So essentially, this particular approach does not address the load distribution across machines as all Pods affinity with each other in a group end up going to the machine with the lowest cost on the connecting arc.



Issue: Also, it is not clear how do we address the following affinity scenario where Pod P1 has affinity with P2 but Pod P2 may not have affinity with P1:

- Pod P1 has affinity with P2 (affinity label value is: P2)
- Pod P2 does not have affinity with P1 (there is no affinity label value in this case)

In this case, incoming Pod P2 will unnecessarily get grouped with Pod P1, even though it does not need to be. Only Pod P1 needs to be grouped with P2 not the other way around. Affinity is not symmetrical.

Appendix

"xor" flow network construct It is often desirable to model in the flow network exclusive disjunctions. These can be useful to express policies that avoid co-locating interfering tasks or that spread tasks across machines/racks for various reasons (e.g., to improve fault tolerance). In

The diagram Figure 5.20a below, I model the “xor” logical operation for two tasks’ flow supply. I connect the tasks to an aggregator node (G) I introduce. I then connect this node only to another node aggregator (O). I set a maximum capacity of one on arc (G; O) to ensure that only one unit of flow can be routed along this arc. Thus, at most one task (i.e., T0,0 or T0,1) will be placed on the machines to which node O connects. In Figure 5.20b, I extend the construct to n tasks. Like previously, I connect task nodes to an aggregator node G that only has one outgoing arc to node O. I set the maximum capacity on this arc to the maximum number of tasks connected to node G that are allowed to be placed. I also set maximum capacities on the outgoing arcs from node O to machine nodes. These capacities control how many tasks are co-located on each machine.

The “xor” network flow construct can be used to express complex conditional preemption constraints such as if task Tx is placed, then task Ty must be preempted as a result. Similarly, the generalised “xor” construct can be used to express conditional constraints, but also complex co-scheduling constraints (such as task anti-affinity constraints) that require several tasks of a job to not share a resource (e.g., rack, machine).