Technical Disclosure Commons

Defensive Publications Series

February 2022

LOAD BALANCING IN ARBITRARILY CLUSTERED MICROSERVICE ARCHITECTURES

Anonymous

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation

Anonymous, "LOAD BALANCING IN ARBITRARILY CLUSTERED MICROSERVICE ARCHITECTURES", Technical Disclosure Commons, (February 28, 2022)
https://www.tdcommons.org/dpubs_series/4940

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

LOAD BALANCING IN ARBITRARILY CLUSTERED MICROSERVICE ARCHITECTURES

ABSTRACT

The present disclosure relates to load balancing in arbitrarily clustered microservice architectures. In cases where the affinity and load-balancing of the servers cannot be controlled or known in advance due to the hierarchical nature of the clustering architecture, measuring typical response times and sorting so the more expensive work is done first (as described in the section above) is a good way for the client or the cluster service to be able to ensure queues and load balancing is as efficient as possible. the present invention allows for arbitrarily deep hierarchies of clustering running with any combination of processors, cores, and threading. It uses the already existing load-balancing mechanisms to allow for efficient queuing and processing across compute resources and attempts to pack the resources as efficiently as possible. It allows for requests in a multi-cluster hierarchy to come in from any layer in the hierarchy and the load balancing between multiple entry points will proceed as needed without additional coordination.

DETAILED DESCRIPTION

Microservice architectures, and particularly ones which use clustering to service clustered requests have a load balancing problem in which the work required to service a bulk request from a client needs to be broken down and assigned to back-end services on processors, but there is no information to know how to effectively balance the load.

In deployments in which a service or database is logically comprised of multiple physical elements, there can be an optimization on how to retrieve data based on cost functions and physical/logical ordering. For example, Apache Spark's Catalyst engine supports logical queries that span multiple physical instances in a cluster. Since the service using this Spark instance contains multiple physical instances, the domain of the service can know the behavior of each element, the cost of getting data from each physical node (local or remote) and how best to optimize and load balance between the physical instances. But in deployments in which multiple microservices are needed to support a generalized cluster architecture, the entry point for the query cannot know details of the physical or logical deployments on the backend of the clustered request. There can be no detailed model of the architecture or cost model needed to drive a load balancing optimization.

The present invention is composed of two parts, needed to form a clustered request in such a way that the backend handlers will be invoked so that the load balancing is predictable and roughly maximally efficient even when a detailed cost model is not possible.

- Measurement of response times for queries and training of clustering service

- Use of measurement data to form clustered messages in such a way to make load-balancing most efficient

In a clustering service, a single request from a client is broken down into multiple internal requests and sent to backend services, possibly on different processors, to service and then the result is combined into a response. These requests are typically coming from an external machine-based client or a pubsub periodic subscription, so the paths and filters provided are usually repeated many times.

The clustering service that breaks the request up into many smaller requests should have a way of measuring on a per request basis

- Overall time
- Transport queue time
- Processing time (user)
- Processing time (real)

These statistics should be

- Stored at the clustering service
- Averaged and tide-marked to reduce noisiness due to varying transport and cpu loads
- Aged to track the most recent response times in the event that the data reported back evolves during the lifetime of the system
- Tagged against the path and filter

Note that since the first clustering service simply breaks the request up to forward to other services, it does not know what processor or core will service those requests. In fact, the services it forwards to may also be clustered services and they may in turn break a request up into smaller requests, or forward to another processor based on high-availability and load-balancing needs in downstream flow.

The other factor is that downstream services may have queuing limits that mean that there are limits to the number of concurrent requests a service can process. If two requests end up in a queue for a downstream services and they are both expensive, the service may serve those requests serially. This is done to prevent overloading CPUs and causing periods of high levels of requests to slow down all requests. It is better to be dedicated to a smaller number of requests in the queue and give responses quickly than try and service a high number of requests in parallel and slow down all responses due to scheduler overhead.

The present invention presumes that only these statistics can be used at a client layer to organize the requests. Because the cluster services can be nested, there is nothing the client can assume about downstream load balancing mechanisms the server can provide (the core assigned to a given request can change multiple times during processing, but the processor assigned to a given request cannot change once the request has been queued). There is no method for a client or clustering service to specify an affinity or anti-

affinity since the physical resource that may execute the operation may be far removed from the resource requesting it.

Once the performance has been measured, it is desired to order the tasks in such a way that will be likely to optimized on the service side. It is important to note that this method of measuring and ordering makes a key assumption that the work being broken down and fed to backend services via a clustering service are

- Independent of each other (one request does not need data from another)
- Completely order independent (one request does not need another request to precede or proceed it

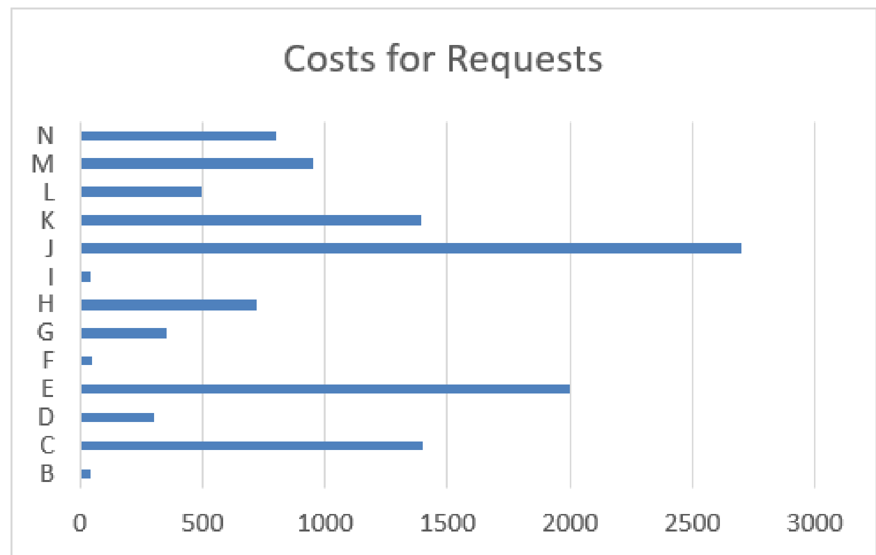
The typical case where this is normally true would be a “get” request for data that spanned multiple backend services. This is the example used to illustrate this concept below.

The main thing to avoid is that when a request is fragmented such that backend services and processors/cores will need to handle more than one request (queued or parallel), that multiple expensive requests will need to be handled by the same service when we could have load-shared it

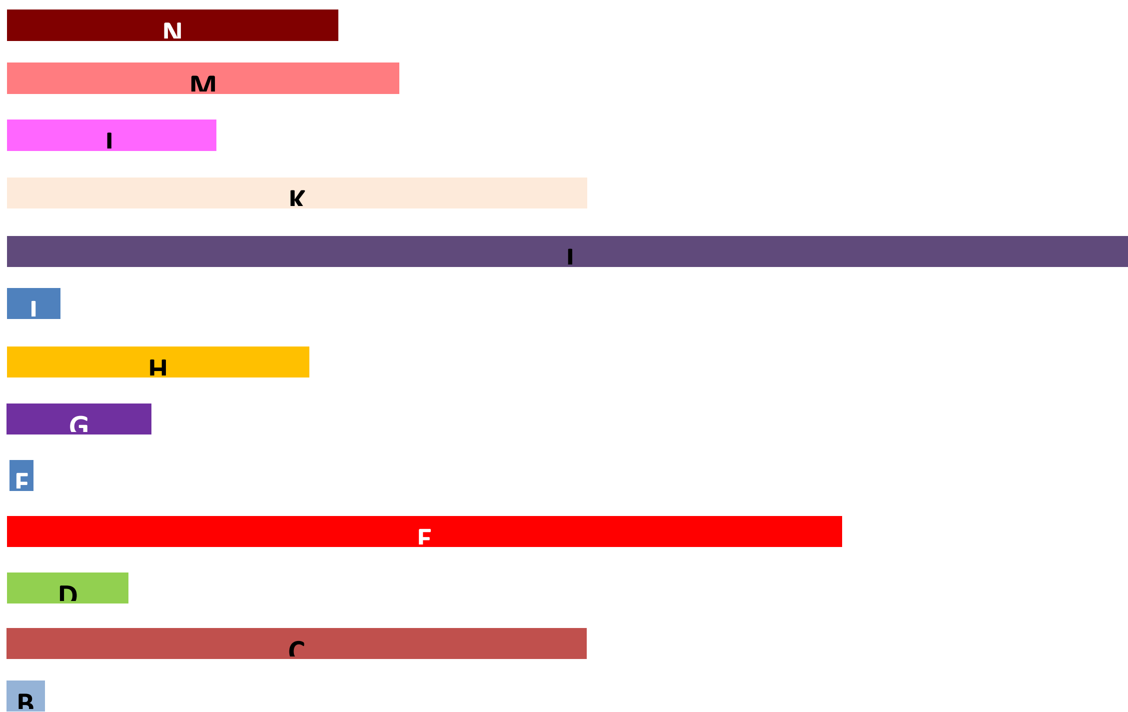
A sample set of requests with random costs that need to be handled by a clustering service is shown in figure 1.

Figure 1

request	cost
B	40
C	1400
D	300
E	2000
F	50
G	350
H	720
I	40
J	2700
K	1390
L	500
M	950
N	800

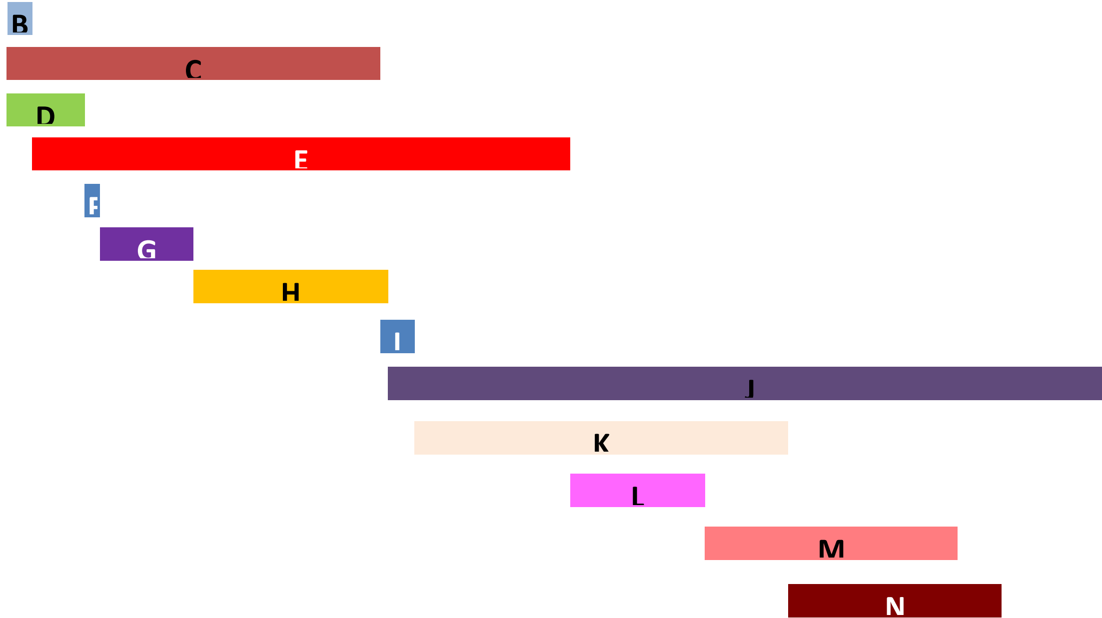


For illustration, compare the work done by the most efficiency load balancing – if a processor is dedicated to each request. All work proceeds in parallel. Total time is 2700 (longest time).

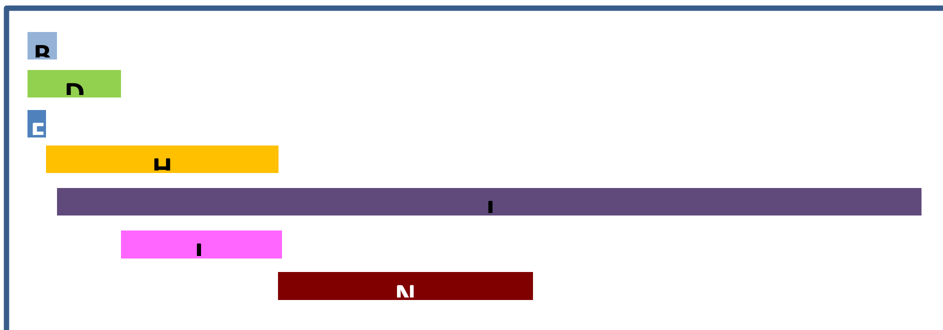


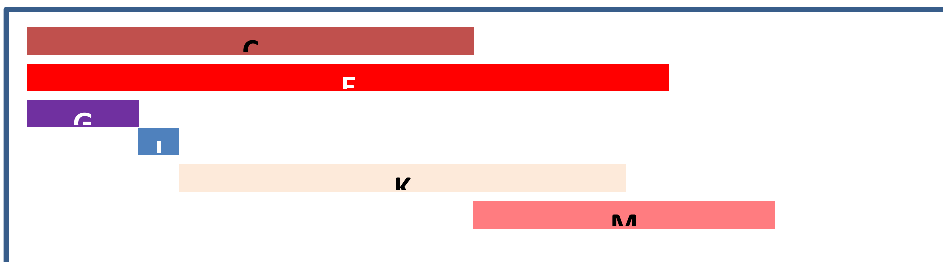
If the requests are all handled by a single service serially with a single or multiple cores, the total time will just be a simple sum of all times required. Total time is 11240 (sum of all times).

If a service is able to service up to three requests at once, and is running on a single processor with multiple cores, the processing may look similar to this with no more than three requests being serviced at once. Total time is 4100 (J+C).



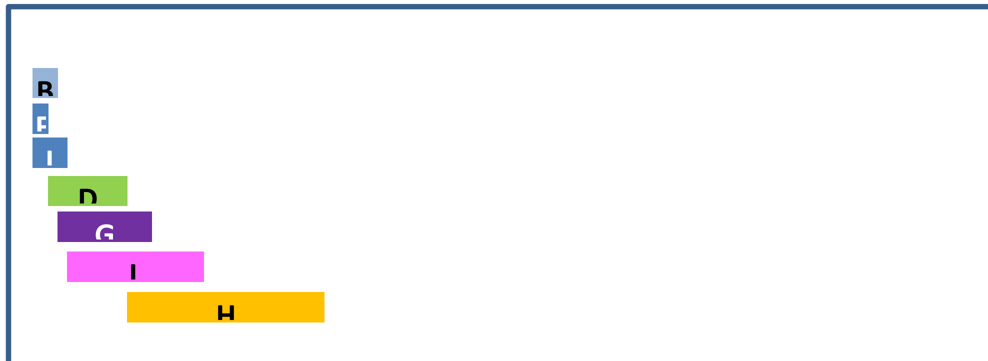
If a service is spread across two processors and each is multi-cored with three threads, the work can proceed in parallel. Total time is 2740 (J+B).

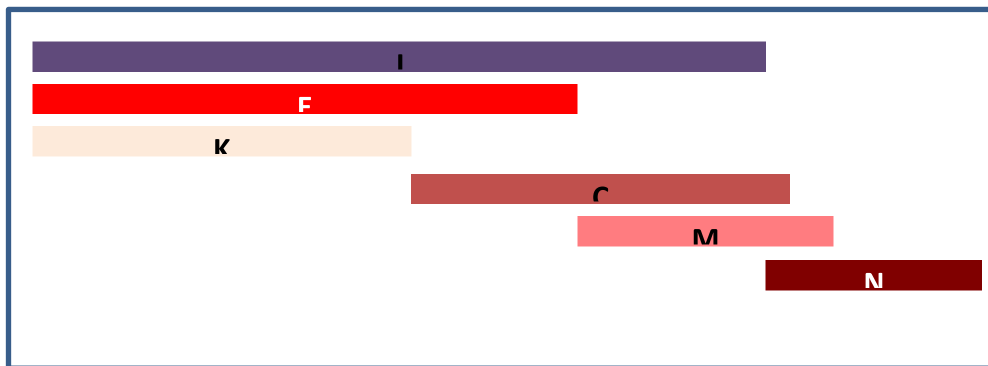




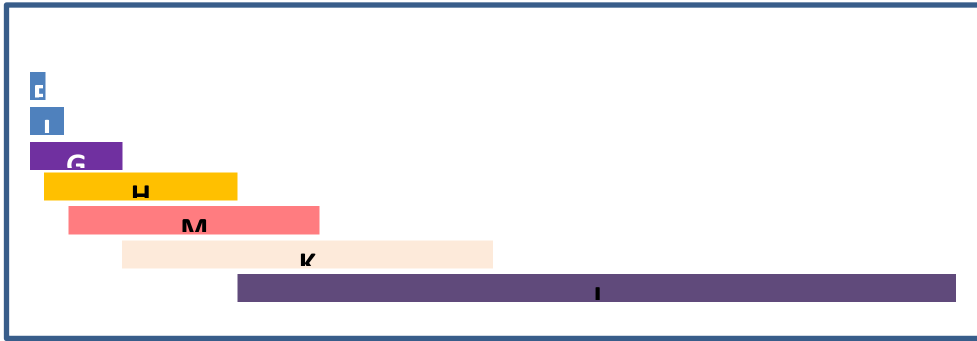
One thing to note is that the last work item on the second processor could have been processed by the first processor since it had a core idle before that work was started, but once the work has been dispatched to a processor, it cannot be dequeued or moved to another member of the cluster. A more drastic version of this is shown below where the random ordering between two processors makes a very inefficient work load.

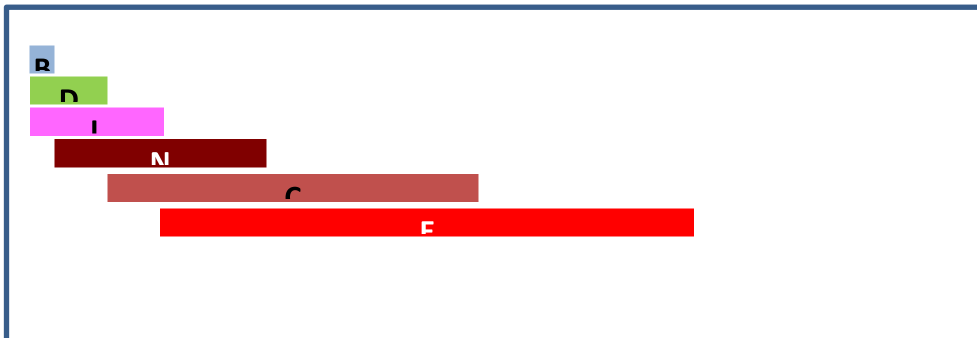
Random ordering of (or unordered) requests are bad because there is no control of how the expensive requests get processed. In this case, the first processor got all the low-cost work and the second processor got all the expensive work. Total time is 3500 (J+N).



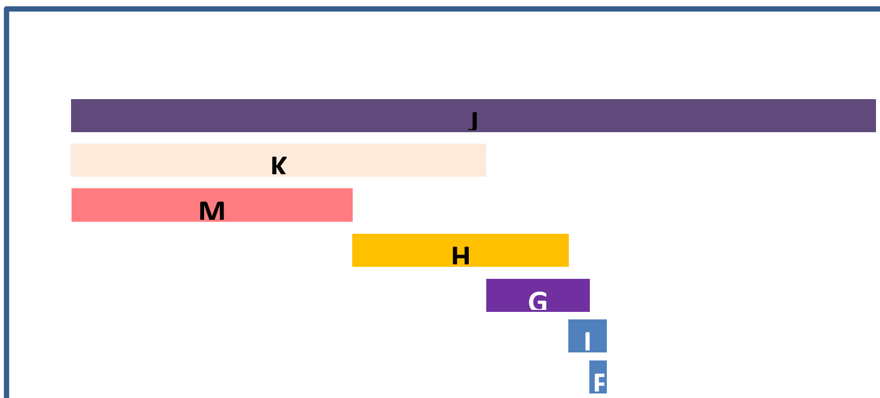


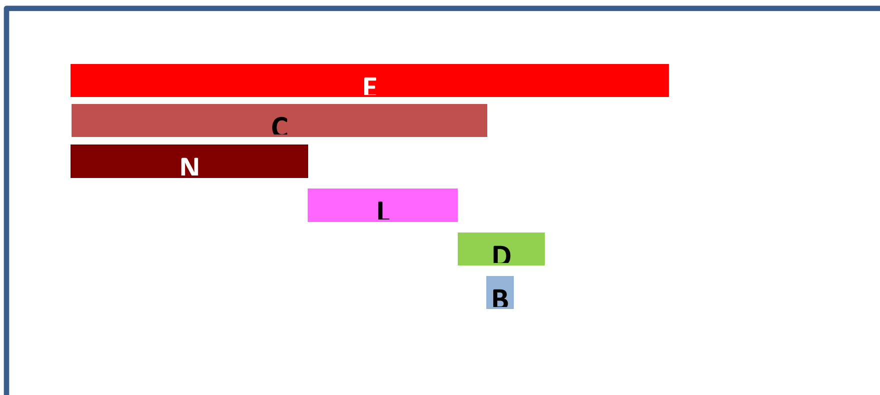
If the work orders are sorted, the load-sharing on the backend will naturally distribute the work to resources and spread similarly expensive items across multiple resources when it can. This is an example of sorting low-cost first in a similar environment. Total time is 3460.





And a highest-cost first. Total time is 2700 (J).





Sorting will allow the backend load balancing and queuing across processors to be as efficient as possible. As another example in table 1, the following costs are assigned to various requests.

Table 1

20	30	40	50	60	70	80
90	100	200	300	400	500	600
700	800	900	1000	1100	1200	1300
1400	1500	1600	1700	1800	1900	2000

When randomized work assignment is done across multiple processors, the timings can vary significantly.

Table 2

# processors	Random			Sorted low to high			Sorted high to low		
	avg	min	max	avg	min	max	avg	min	max
1	24	24	24	24	24	24	24	24	24
2	12	12	12	12.3	12.2	12.3	12.3	12.2	12.3
3	8.98	7.6	11	8.1	8.1	8.1	8.1	8.1	8.1
4	7.14	5.4	8.4	6.63	6.5	6.8	6.45	6.4	6.5
5	8.1	6.5	10	6.2	6.2	6.2	7.6	7.6	7.6
6	7.55	5.9	9.5	6.63	6.6	6.7	6.6	6.6	6.6
7	6.98	5.4	8.1	6.05	6	6.1	5.8	5.8	5.8
8	6.51	5.5	8.9	5.35	5.3	5.5	5.23	5.1	5.3

Figure 2

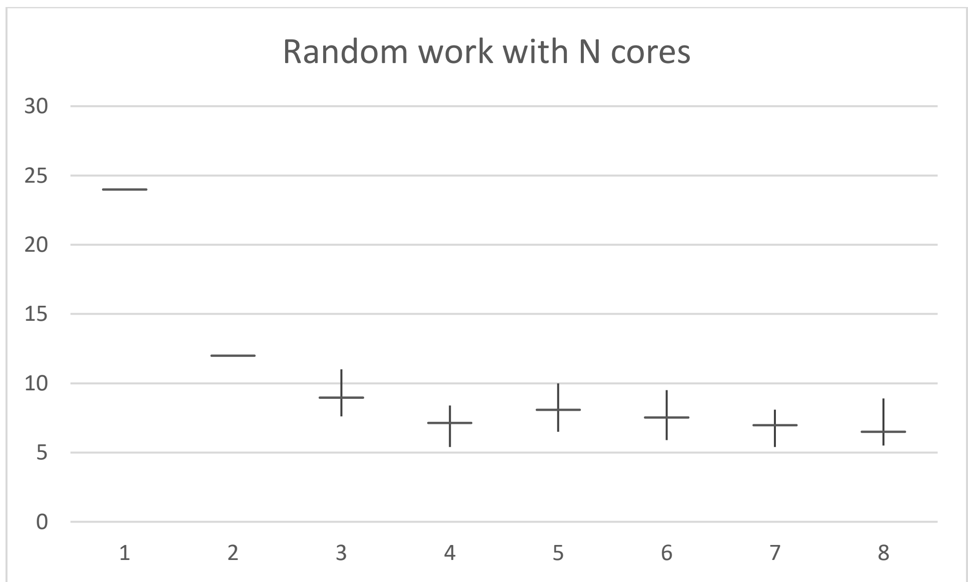


Figure 3

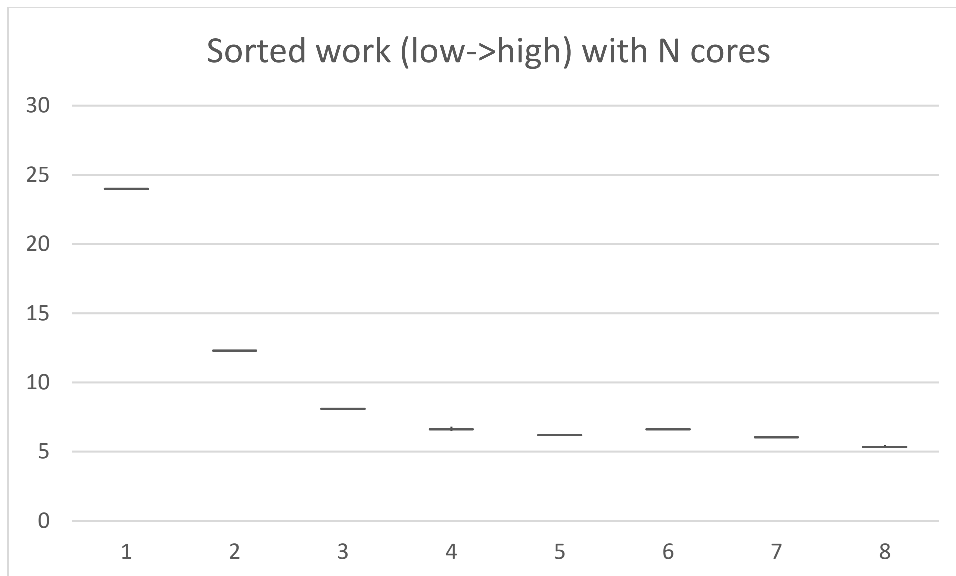
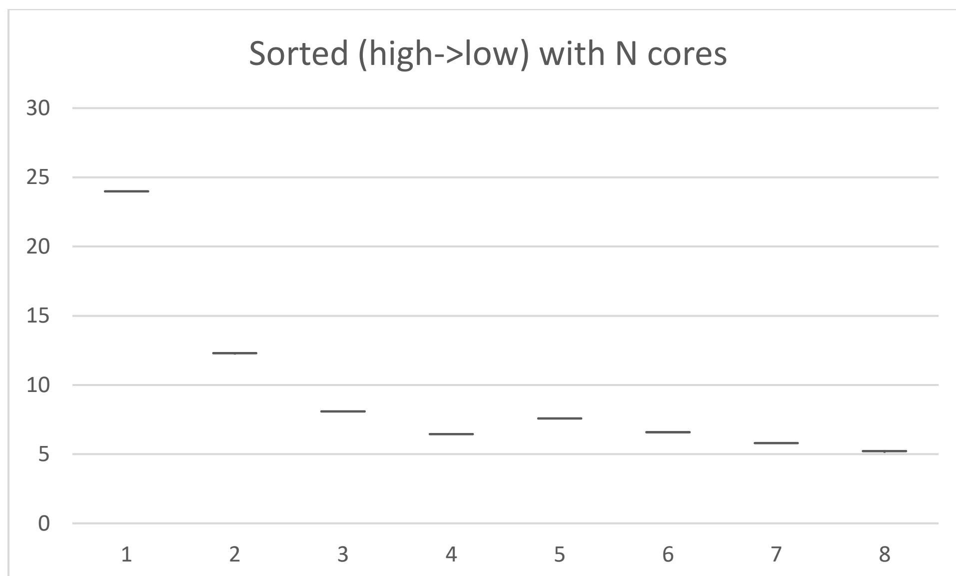


Figure 4



Sorted work order shows much more consistent and lower times. The highest-cost-first sorting has some advantages.

- It starts with the slowest requests and will fill in the processors with the most work the soonest. In this example, the total time needed is limited by the highest cost request. And the second processor is done its work before the second-highest request is done

- When there is uncertainty about the costs and the sort is not accurate, getting the potentially highest cost work done first will still allow the extra processing time to catch up if it can be done sooner
- It is best to have the processors loaded up with as much work as possible up front so the server side load-balancing can fairly give work to other processors or cores. Starting from the lowest cost may end up giving more work requests to other queues since the work items may be so small the queues may be changing as the work is completed.

In cases where the affinity and load-balancing of the servers cannot be controlled or known in advance due to the hierarchical nature of the clustering architecture, measuring typical response times and sorting so the more expensive work is done first (as described in the section above) is a good way for the client or the cluster service to be able to ensure queues and load balancing is as efficient as possible.

A processor will load-balance by moving threads between processors as work is completed, but in a cluster, once work is assigned to a service on a processor, it is queued and cannot be unassigned if other compute resources free up. Other approaches may keep the queuing mechanism at a single location and assign work items only as processor and service queues free up.

Other approaches require a detailed knowledge of the backend physical layout and cost functions in order to drive load balancing from the cluster service. For most installations, custom code is needed on the backend in order for an optimizer to control the load balancing. This increases the scope of the cluster microservice to essentially consume all backend services into one holistic service since the control and operation of the backend must be treated as a single service.

The reason this can work with a single service (like Catalyst) is that this is an intra-service optimization for a service that is spread over multiple physical instances, all elements of the same Spark instance. The invention tries to solve the case where a query from a service is cascaded to a set of independent backend services that are in a cluster, so load balancing is needed between services that don't share a common infrastructure or have the knowledge of the backend implementation to properly know how to do a load balancing optimization. This is essentially optimization between multiple independent black box implementations. Since we can't see inside, we need to probe and measure so we can predict and order the request. This may not always be as efficient as if the services were combined so the single service could be optimized but if independent services are needed, this can provide some optimization and consistency.

Managing the clustering queues from a single location means that clustering can only happen from one location. In hierarchical clustering, this is not a good option, since requests may come in at multiple levels of a clustering hierarchy. A central load-balancer cannot accommodate requests coming in from different locations.

Other solutions which do load-balancing across processors will many times only support one level of clustering. This is a rigid architecture and does not allow the flexibility to install and manage higher levels of clustering and load balancing without rearchitecting the communications between the services.

Solutions which need a detailed model to drive backend load balancing and optimization break a microservice model and force the clustering to all be done by one service that can model and control the optimization in a central fashion. This is not desired for reasons of maintenance and code sharing since it forces architectural choices and larger services where this would not be normally desired.

This invention allows for arbitrarily deep hierarchies of clustering running with any combination of processors, cores, and threading. It uses the already existing load-balancing mechanisms to allow for efficient queuing and processing across compute resources and attempts to pack the resources as efficiently as possible. It allows for requests in a multi-cluster hierarchy to come in from any layer in the hierarchy and the load balancing between multiple entry points will proceed as needed without additional coordination. This is not the most efficient way of load balancing. Nothing can predict how concurrent requests will collide, or if a given request will take longer or shorter to process in a given iteration. When affinity and queue management cannot be directly controlled by the client or the cluster service (which would be complex operation to expose and manage), a cost-first sorted approach to work assignment can offer consistent improvements in efficiency.

It will be appreciated that some embodiments described herein may include one or more generic or specialized processors (“one or more processors”) such as microprocessors, digital signal processors, customized processors, and Field-Programmable Gate Arrays (FPGAs) and unique stored program instructions (including both software and firmware) that control the one or more processors to implement, in conjunction with certain non-processor circuits, some, most, or all of the functions of the methods and/or systems described herein. Alternatively, some or all functions may be implemented by a state machine that has no stored program instructions, or in one or more Application-Specific Integrated Circuits (ASICs), in which each function or some combinations of certain of the functions are implemented as custom logic. Of course, a combination of the aforementioned approaches may be used. Moreover, some embodiments may be implemented as a non-transitory computer-readable storage medium having computer-readable code stored thereon for programming a computer, server, appliance, device, etc. each of which may include a processor to perform methods as described and claimed herein. Examples of such computer-readable storage mediums include, but are not limited to, a hard disk, an optical storage device, a magnetic storage device, a ROM (Read Only Memory), a PROM (Programmable Read-Only Memory), an EPROM (Erasable Programmable Read-Only Memory), an EEPROM (Electrically Erasable Programmable Read-Only Memory), Flash memory, and the like. When stored in the non-transitory computer-readable medium, the software can include instructions executable by a processor that, in response to such execution, cause a processor or any other circuitry to perform a set of operations, steps, methods, processes, algorithms, etc.

Although the present disclosure has been illustrated and described herein with reference to preferred embodiments and specific examples thereof, it will be readily apparent to those of ordinary skill in the art that other embodiments and examples may perform similar functions and/or achieve like results. All such equivalent embodiments and examples are within the spirit and scope of the present disclosure.