VMware vSphere 6.7 Tagging Best Practices

Performance Study - August 17, 2020

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

Table of Contents

1 Executive summary

Writing code to use vSphere tags can be challenging in large-scale environments. In this whitepaper, we discuss the scalability limits of tags, and we give some tips and tricks for writing performant tagging code in either Java or PowerShell.

2 Introduction

vSphere 5.1 introduced an inventory tagging feature that has been available in all later versions of vSphere, including vSphere 6.7. Tags let datacenter administrators organize different vSphere objects like datastores, virtual machines, hosts, and so on. This makes it easier to sort and search for objects that share a tag, among other things.

In vSphere, tags are organized into categories. A category is typically used for a high-level description. For example, a category might be "OS type" or "application name." Within a category, the tags are the distinct values for that category. For example, if the category is "OS type," then the tags might be "Linux" or "Windows 2016." If the category is "application name," the tags might be "DB," "Middleware," or "vCenter Server." If the category is "CPU type," the tag values might be "AMD," "Intel Broadwell," or "Intel Cascade Lake."

Writing code to use tags can be challenging in large-scale environments. In this whitepaper, we discuss the scalability limits of tags, and we give some tips and tricks for writing performant tagging code in either Java or PowerShell. This paper complements a recent VMware Performance team blog on writing performant code using PowerShell [1], which focused exclusively on PowerCLI cmdlets vs. direct tagging service calls. While the performance results in this paper are specific to the hardware configuration and VCSA configuration in our lab, the basic trends will be the same as in customer environments.

We first discuss the scalability limits for tags in vSphere 6.7. We then provide best practices for writing code for tagging.

3 Terminology and Scope of Paper

Before describing the scale limits for tagging, we first define some terms:

- "Category definitions" refers to the number of tag categories, for example, "OS type" or "application name."
- "Tag definitions" refers to the number of tag definitions in the system, for example, "Windows" from category "OS type" or "SQL Server" under category "application name."
- "Tag associations" refers to the mapping between a tag and an object (like a virtual machine or datastore). For example, if tag "Windows" is applied to VM "vm-111," then that is one association, and if tag "Windows" and tag "SQL Server" are both applied to "vm-111," then that is two associations. If a user associated 25 tags ("Windows," "SQL Server," "Pasadena Datacenter", etc.) with 1,000 VMs, that would be 25,000 tag associations.

There are no *hard* limits to the number of tags, categories, and tag associations. However, increasing numbers of tags, categories, or associations can impact performance. In this paper, we have tried to provide information that you can use to understand the performance limits in your environment. We give best practices for how to write scripts to create tag associations and retrieve association information from vCenter.

We will focus on Java- and PowerCLI-based programs, although the same principles apply to other languages like Python. Moreover, we will focus primarily on three operations:

1. Creating tag definitions
2. Associating (attaching) tags to entities like VMs or datastores
3. Retrieving tag associations

These three operations are the most common tasks performed by scripts or plugins.

4 Tagging APIs: Java

We will first discuss tag creation. In vSphere, a tag can only be created as part of an existing category. Thus, a category must be created before a tag can be added to it. The performance considerations for creating categories are similar to those of creating tags, so we discuss only tag creation in this section. For a simple example of creating a category using Java, please see "Example J2: Create Category" in the "Appendix."

Resources:

- The Java API for vSphere 6.5 is documented here [2].
- The Java API for vSphere 6.7 is documented here [3] and here [4].

4.1 Create tag (assuming category is already created)

Sample code for creating a category and creating a tag within that category is given in the "Appendix" (please see "Example J2: Create Category" and "Example J3: Create Tag").

Tag creation requires a name, a description, and a cardinality. The list of associable types (that is, what types of objects, like VMs or hosts, can be associated with this tag is optional: if it is not supplied, then a tag can be associated with any object. A description of cardinality is given in the Performance team tagging blog [1], but briefly, it means that multiple tags (say, "Alice" and "Bob") from a given category (say, "Owners") can be assigned to a given object (like a VM).

Creating a tag definition is a serial operation: only one tag can be created at a time. The time to create a tag varies with the number of tags per category. For example, creating many tags under a single category is slower than creating a smaller number of tags in multiple categories.

Figure 1 shows the latency to create 8,000 tag definitions under a single category. For example, if the category name were "Application," the tag definitions might be "SQL Server," "SAP Hana," "Oracle," "Kafka Messaging," etc. The time increases linearly with the number of tags when all tags are under the same category. As the figure shows, creating the first tag takes around 4 milliseconds (ms), and creating the last tag takes 60 ms.

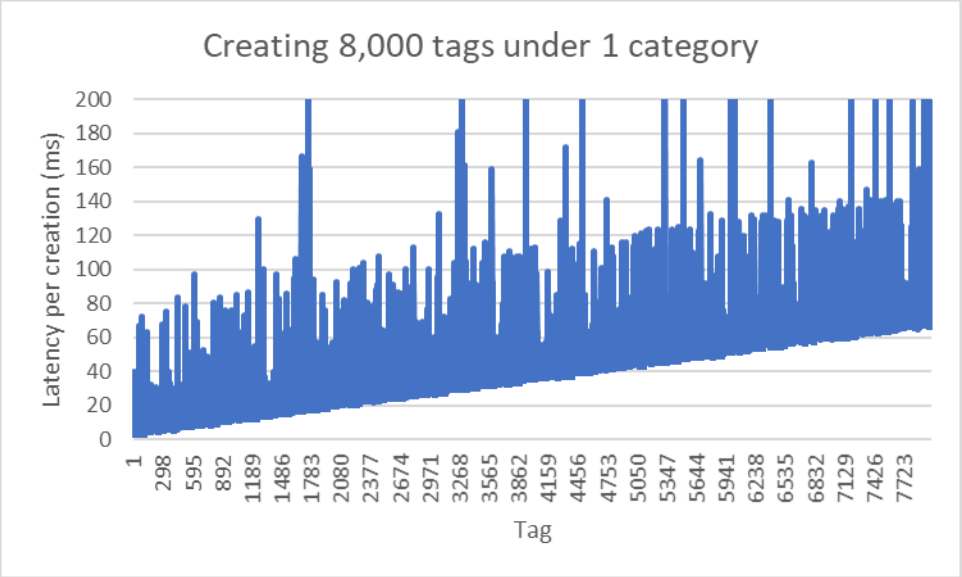

Figure 1: Creating 8,000 tags under 1 category. The time to create a tag increases linearly with the number of tags.

Figure 2 shows the latency to create 8,000 tag definitions under 200 categories. In each case, there are 40 tag definitions per category. As the graph indicates, the latency to create tags increases much more slowly with fewer tags per category: the time to create the first few tags is 4-16 ms, and the latency to create the last few tags is around 16-20 ms. *Because the performance is robust with this configuration, if practical, we recommend using multiple categories with fewer tags per category vs. a small number of categories with many tags in them.*

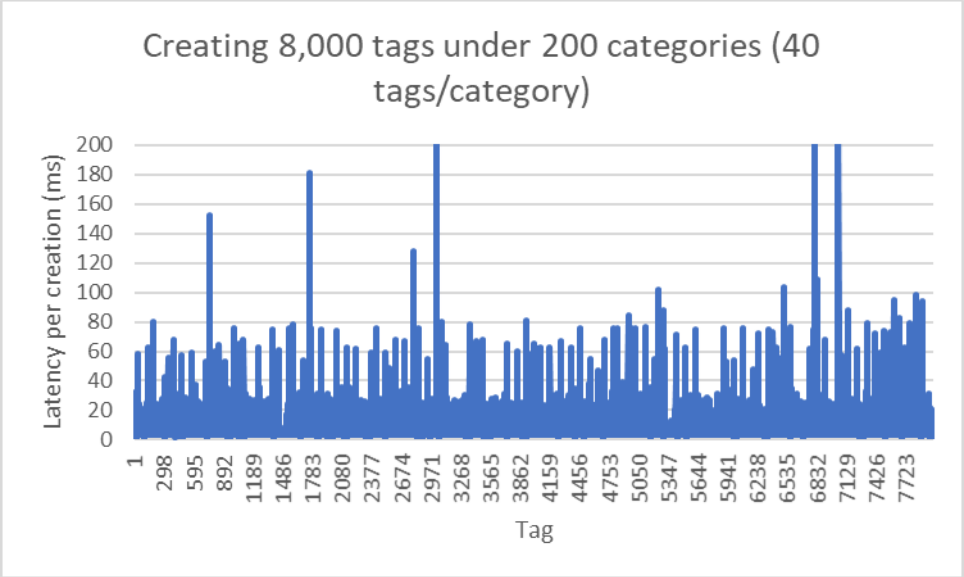

Figure 2: Creating 8,000 tags under 200 categories. Spreading tags among categories speeds up tag creation. This makes it roughly constant for small numbers of tags per category.

One other option to speed up tag creation is to use multiple clients to create tags concurrently. In our labs, we have tested up to 8 clients, with promising speedups (up to 7x for 8 clients).

4.1.1 Caveat: Tombstones when creating and destroying tags

If you create and then destroy a lot of tags, the tagging service accumulates "tombstones" (residual information in the tag database). These tombstones add latency overhead when creating more tags. This happened in our setups after creating and then destroying 3,000 tags. In our case, initial tag creations took tens of milliseconds. After we had created and then destroyed 3,000 tags, the latency increased sharply. One way to detect this is to measure tag creation latency. The typical latency should be tens of milliseconds. Suppose you have fewer than 8,000 tags, but you have created and destroyed a lot of tags. If the latency to create your tag is over 100 ms, and if you currently have fewer than 8,000 tags in the system, you may be experiencing tombstone effects due to the destroyed tags. Over time, as more and more tombstones are created, the overhead of these tombstones increases, and the latency to create tags increases. To diagnose and resolve this issue, please refer to KB 52387 [5].

In the next section, we will talk about tag associations.

4.2 Associate tag(s) with VM(s)

There are 3 ways to attach a tag to a VM:

1. `attach(tagID, VM_ID)`
2. `attachTagToMultipleObjects(tagID, list of VM_IDs)`
3. `attachMultipleTagsToObject(list of tagIDs, VMID)`

Each of these calls is useful in different scenarios. The main difference is how many calls to the tagging service are required.

4.2.1 Single tag to single VM

When attaching a single tag to a single VM, it is easiest to use a simple `attach()` call (option 1 above). Please see Example J4: Associating a tag with a VM (attach())in the "Appendix", where we assign a tag "Windows 10" from category "GuestOS" to a VM. We recommend keeping a map of tag ID to tag name and VM ID to VM name, if possible, in order to avoid having to call vCenter or the tagging service each time such a mapping is needed. Pseudocode for using `attach()` is given here:

```
1. Get the tag ID (tagID) from tag name.
2. Get the VM ID (VM_ID) from the VM name.
3. attach(tagID, VM_ID).
```

Attaching a single tag to a single VM typically takes around 15 ms, but the time varies as the number of tag associations increases. Figure 3 shows the latency to attach a tag to a VM as we increase the number of VMs from 1 to 10,000. The first few `attach()` calls take around 15 ms. The latency varies, but most of the `attach()` calls take around 15 ms, while the last one takes around 17 ms. The total end-to-end time is 3 minutes.

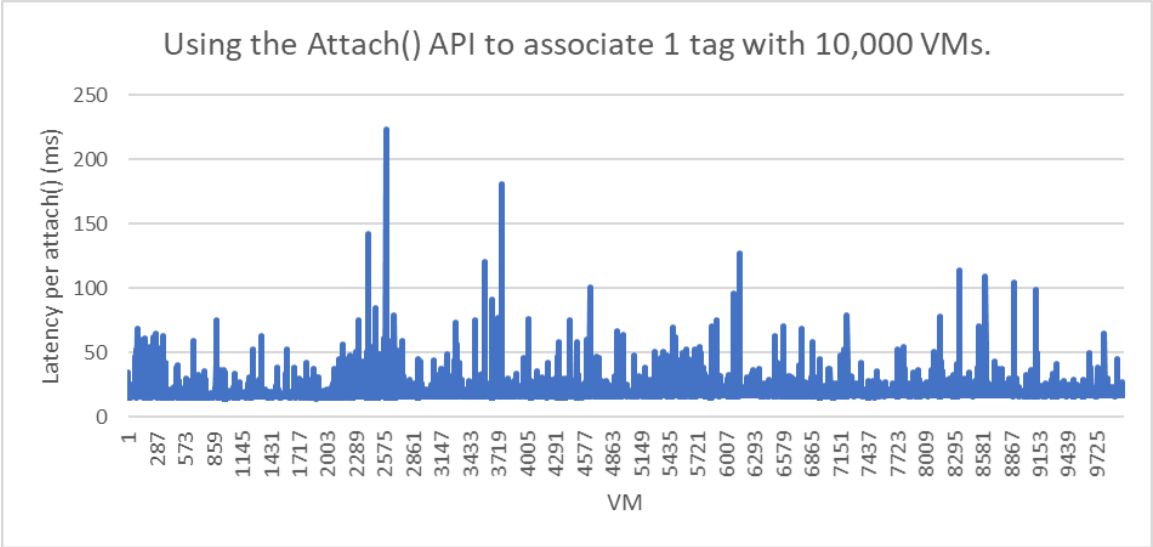

Figure 3: Using `attach()` to associate 1 tag with each of 10,000 VMs. The time for an individual `attach()` increases slightly with the number of VMs.

4.2.2 Single tag to multiple VMs

When attaching a single tag to multiple VMs, it is easiest to use option 2, `attachTagToMultipleObjects()`. This call amortizes the cost of attaching the tag over multiple VMs. For attaching the same tag to 10 VMs (VMID_ 0-VMID_ 9), only one call to the tagging service is required, as opposed to 10 if we had used an `attach(tagID,VMID_N)` call in a loop over each VM. See "Example J5: Associating a tag with multiple VMs (attachTagToMultipleObjects())" in the "Appendix" for an example of attaching a single tag to multiple VMs using `attachTagToMultipleObjects()`. Pseudocode is given here:

```
1. Get tag ID (String tagID)
2. Get list of VM Dynamic IDs (List<DynamicID> VM_IDs)
3. attachTagToMultipleObjects(tagID,VM_IDs)
```

Attaching a single tag to multiple VMs using `attachTagToMultipleObjects()` is more efficient than looping over every VM and calling `attach()`. Table 1 compares the latency of attaching a single tag to multiple VMs using `attachTagToMultipleObjects()` vs. using `attach()`. As noted in the caveat above, in the case of `attachTagToMultipleObjects()`, we attach the tag to 2,000 VMs at a time. In the case of `attach()`, we sequentially attach the tag to each VM.

Operation	Number of iterations	Latency for 1 tag, 20,000 VMs
`attach()`	20,000 (1 per VM)	372 seconds
`attachTagToMultipleObjects()`	10 (2,000 VMs at a time)	231 seconds

Table 1: attach() vs. attachTagToMultipleObjects()

4.2.2.1 Chunking VMs in calls to attachTagToMultipleObjects()

We suggest putting no more than 2,000 VMs per `attachTagToMultipleObjects()`. Exceeding this number may cause errors, since the tagging service has some limits regarding the number of items in a request. If using Java, you will see this sort of error in the log file for the vapi-endpoint service (that is, /var/log/vmware/vapi/endpoint/endpoint.log):

```
Received request (910,429 bytes) is bigger than the allowed request size (204,800 bytes) -
request blocked!
```

4.2.3 Multiple tags to a single VM

When attaching multiple tags to a single VM, it is best to use option 3, `attachMultipleTagsToObject()`. We can attach 10 tags (tag_0-tag_9) to a single VM with one tagging service call, instead of the 10 that would be required if we used attach(tag_N,VMID) in a loop over each tag. We give an example of attaching multiple tags to VMs in the "Appendix" in "Example J6: Associating multiple tags with multiple VMs (attachMultipleTagsToObject())". Here is pseudocode to accomplish this:

```
1. Get tag IDs and put them into a list (List<String> tagIDs)
2. Get VM ID (VM_ID)
3. attachMultipleTagsToObject(VM_ID,tagIDs)
```

For best performance, we recommend using no more than 2,000 tags in a single call.

Figure 4 shows the latency of attaching 1 through 200 tags to a single VM using `attachMultipleTagsToObject()`. The latency is nearly constant until around 50 tags, and then it starts to grow linearly.

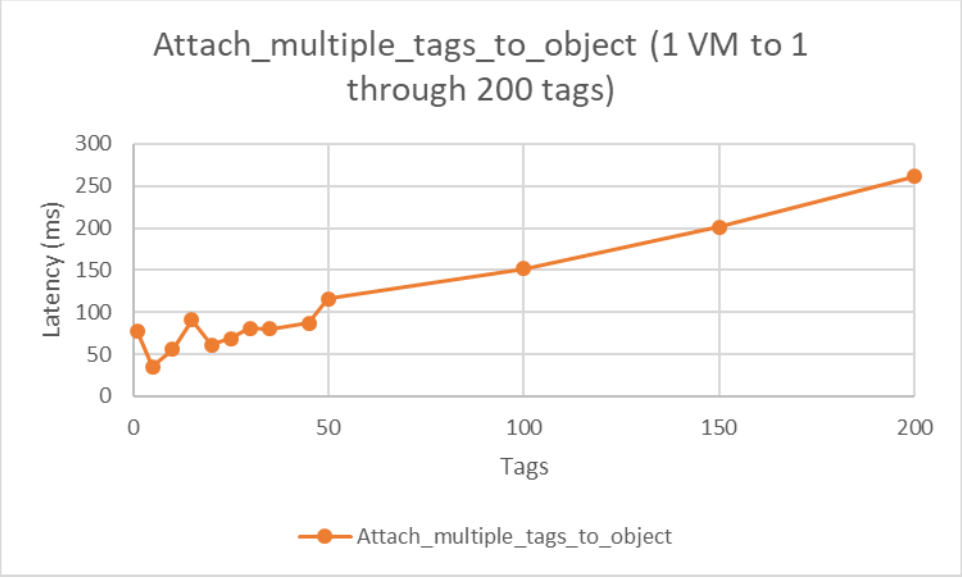

Figure 4: `attachMultipleTagsToObject()` for 1-200 tags and 1 VM. The latency is fairly steady until around 50 tags, when it starts to grow linearly.

In figure 5, we compare the latency of attaching multiple tags (1 up to 2,000) to a single VM using `attachMultipleTagsToObject()` vs. using serial `attach()` calls vs. `attachTagToMultipleObject()`. As the figure shows, each method works well with a small number of tags, but `attachMultipleTagsToObject()` works best as the number of tags grows. For a small number of tags (< 20), the latency of `attachMultipleTagsToObject()` is around 80 ms. For 2,000 tags, the latency is around 2.6 seconds (2600 ms).

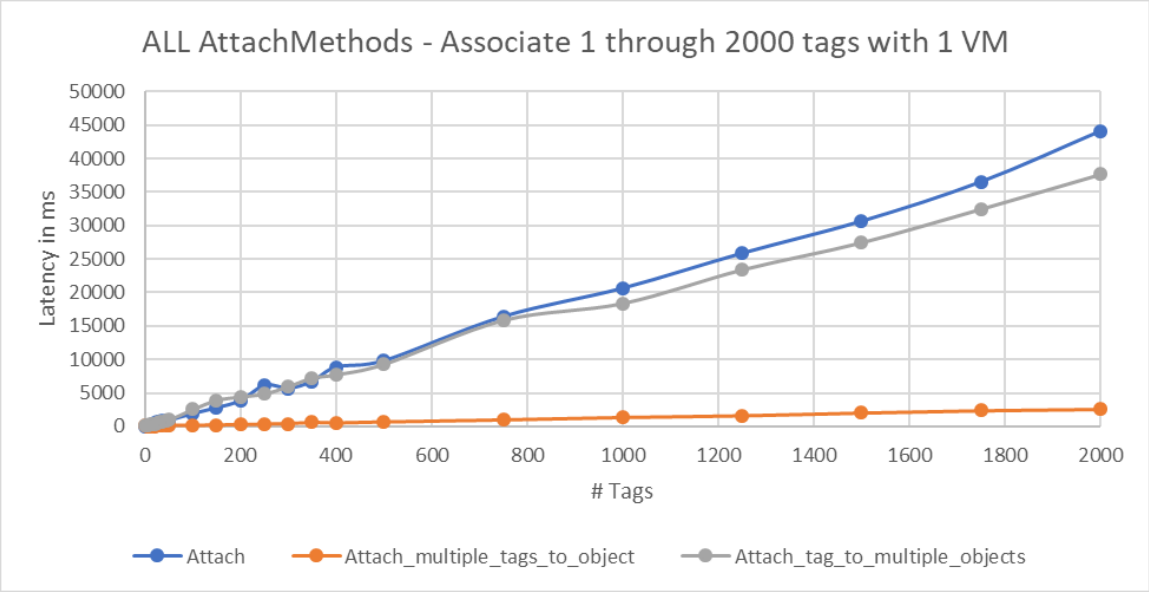

Figure 5: Attaching 1 through 2,000 tags to 1 VM. `attachMultipleTagsToObject()` is well-suited for this use case.

4.2.4 Multiple tags to multiple VMs

The previous cases considered assigning one tag to multiple VMs or multiple tags to one VM. In this section, we discuss assigning multiple tags to multiple VMs.

4.2.4.1 10 tags associated with 1,000 VMs (10,000 tag associations)

Suppose we had a group of 10 tags (for example, "Windows 2019 Server," "DB app," etc.), and we wanted to attach them to a group of 1,000 VMs (for example, VM_0, VM_1, …, VM_999). As the preceding discussion suggests, there are two ways to accomplish this:

1. `attachTagToMultipleObjects()` Pseudocode

```
Get tag IDs → List<String> tagIDs
Get VM IDs → List<DynamicId> VM_IDs
for (String tagID : tagIDs) {
     attachTagToMultipleObjects(tagID, VM_IDs)
}
```

2. `attachMultipleTagsToObject()` Pseudocode

```
Get tag IDs → List<String> tagIDs
Get VM IDs → List<DynamicId> VM_IDs
for (DynamicID VM_ID : VM_IDs) {
     attachMultipleTagsToObject(VM_ID, tagIDs)
}
```

We give more complete code snippets in the "Appendix."

In table 2 below, we show the relative performance of these two calls for assigning 10 tags to 1,000 VMs from our internal testing labs.

API Call	Number of iterations	End-to-end latency (seconds)	Time per iteration
`attachTagToMultipleObjects()`	10	147	15 s
`attachMultipleTagsToObject()`	1,000	30	30 ms

Table 2: Latency for assigning 10 tags to 1,000 VMs (a total of 10,000 associations). The end-to-end latency depends on the per-iteration cost.

As the table indicates, for this modest number of tags and VMs, `attachMultipleTagsToObject()` is faster than `attachTagToMultipleObjects()`, even though `attachMultipleTagsToObject()` executes more iterations. In the first case, each iteration is creating 1,000 tag associations (that is, 1 tag is being associated with 1,000 VMs). In the second case, each iteration is creating 10 associations (that is, associating 10 tags to 1 VM). The performance difference is due to calls to the tag association database, and we are working in our labs to improve this for future releases.

4.2.4.2 15 tags associated with 5,000 VMs (75,000 tag associations)

In the previous example, the number of associations (10,000) was small enough that the latency to create each association was constant. However, the latency to attach tags to VMs increases as the number of associations increases.

4.2.4.2.1 attachTagToMultipleObjects()

Figure 6 shows the latency of attaching multiple (1, 5, 10, 15) tags to up to 5,000 VMs using `attachTagToMultipleObjects()`. This creates up to 75,000 tag associations. To conform to the limit of 2,000 objects per call, for this example, we divided the 5,000 VMs into 2 batches of 2,000 VMs and 1 batch of 1,000 VMs. We then looped through the tags and attached the tag to each batch. Here is some pseudocode to illustrate this:

```
// (not shown): Get tag IDs into List<String> tagIDs
// (not shown): Get VM IDs into List<DynamicId> VM_IDs
int chunkSize = 2000
int numChunks = ((VM_IDs.size()/2000) + 1)
for (String tagID : tagIDs) {
   for (int j = 0; j < numChunks; j++) {
      int index = j * chunkSize
      // Make sure we don't exceed the bounds of the VM ID list
      end = max(VM_IDs.size(),index+chunkSize);
      attachTagToMultipleObjects(tag, VM_IDs.sublist(index, end));
   }
}
```

In the graph below, the time to attach the first tag to all VMs is around 100 seconds, while the time to attach the 15th tag to all VMs is around 200 seconds. The end-to-end time to attach all 15 tags is the sum of the time for each iteration, around 2,200 seconds. This creates 75,000 associations. It is important to note that this is above the out-of-the-box suggestion of 25,000 associations (see "Scale Numbers for Good Performance" in section 8), but we are can achieve this using the API by using batches of VMs. While the API works properly, the UI may be slow in this case.

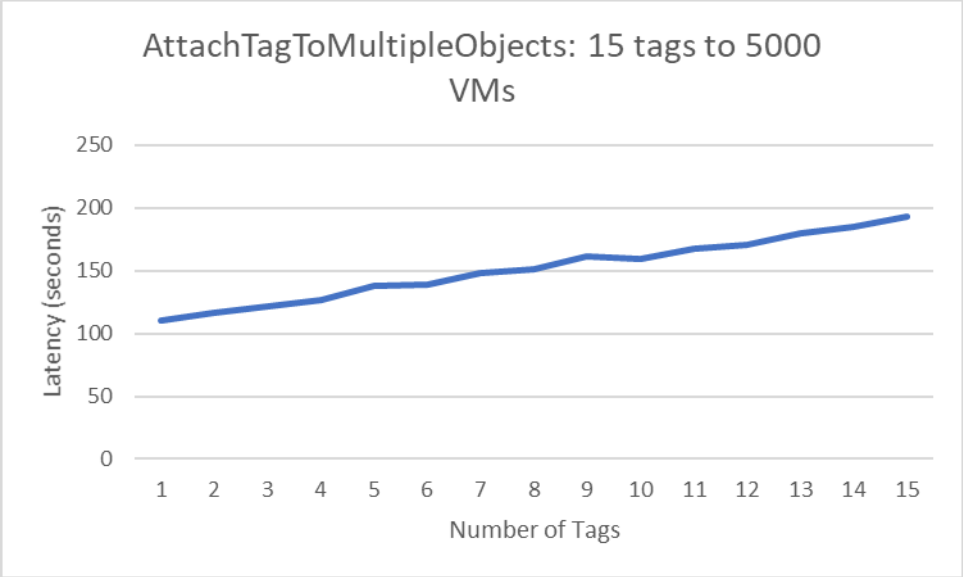

Figure 6: attachTagToMultipleObjects(): 15 tags to 5,000 VMs.

4.2.4.2.2 attachMultipleTagsToObject()

We can attach 15 tags to 5,000 VMs using `attachMultipleTagsToObject()` instead. The results are in figure 7. To use `attachMultipleTagsToObject()`, we loop over VMs one at a time and attach 15 tags at once:

```
// (not shown) Get tagIDs into List<String> tagIDs
// (not shown) Get VM IDs into List<DynamicId> VM_IDs
for (DynamicID VM_ID : VM_IDs) {
   attachMultipleTagsToObject(VM_ID, tagIDs)
}
```

As figure 7 indicates, the time to attach 15 tags to the 1st VM is around 40 ms, and the time to attach 15 tags to the 5,000th VM is around 50 ms. The total time to attach 15 tags to 5,000 VMs is the sum of the latency of each iteration, around 266 seconds.

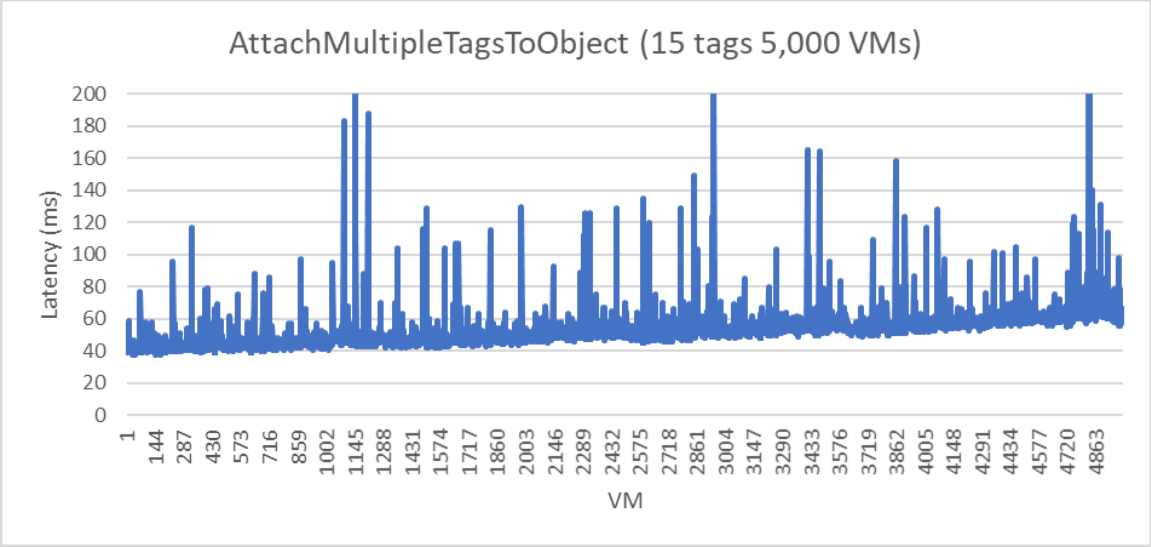

Figure 7: `attachMultipleTagsToObject()`. This call scales roughly linearly with the number of objects.

Caveat about number of objects or tags in a single call. As mentioned earlier, while it is faster to use these batched `attachTagToMultipleObjects()` and `attachMultipleTagsToObject()` calls, there is a limit to the size of a request that can be processed by the tagging service. When this limit is exceeded, the request cannot be processed. For `attachTagToMultipleObjects()`, to avoid hitting this limit, we recommend using no more than 2,000 VMs within a single call. For `attachMultipleTagsToObject()`, we recommend no more than 20 tags per call. If you want to associate a single tag with 20,000 VMs, you should do it using 10 iterations with 2,000 VMs per iteration.

In table 3, we summarize the performance of these API calls when attaching 15 tags to 5,000 VMs (75,000 tag associations):

API Call	Number of Iterations	End-to-end latency
`attachTagToMultipleObjects()`	45 (15 tags * 3 batches)	37.8 minutes
`attachMultipleTagsToObject()`	5,000 (one per VM)	4.4 minutes

Table 3: Latency for assigning 15 tags to 5,000 VMs (a total of 75,000 associations). The end-to-end latency depends on the per-association cost for attachTagToMultipleObjects() and the per-iteration cost for attachMultipleTagsToObject().

As the table indicates, `attachMultipleTagsToObject()` is faster with a large number of VMs.

Table 4 shows a few more data points: 10 tags associated with 1,000 VMs, and 1, 10, or 15 (hereafter referred to as {1,10,15}) tags associated with 5,000 VMs. For large numbers of VMs and more than one tag, `attachMultipleTagsToObject()` scales the best. With one tag and many VMs, `attachTagToMultipleObjects()` scales the best.

Operation	10 tags to 1000 VMs (10k associations)	1 tag to 5,000 VMs (5K associations)	10 tags to 5,000 VMs (50K associations)	15 tags to 5,000 VMs (75K associations)
`attach()`	3.6 minutes	2.1 minutes	26 minutes	48 minutes
`attachTagToMultipleObjects()`	2.5 minutes	1.2 minutes	20 minutes	38 minutes
`attachMultipleTagsToObject()`	0.5 minutes	1.5 minutes	3.4 minutes	4.4 minutes

Table 4: End-to-end latency to attach tags to VMs. We show 10 tags associated with 1,000 VMs, and {1,10,15} tags associated with 5,000 VMs. `attachMultipleTagsToObject()` scales the best when the number of VMs is large.

The *takeaway* points from the table above are as follows:

1. The time to perform a single association increases as more tag associations are added.
2. Batching APIs (`attachTagToMultipleObjects()` or `attachMultipleTagsToObject()`) are preferable to looping over every tag and every VM and using an `attach()` on each tag/VM pair.
3. When performing an operation on more than 2,000 VMs or 2,000 tags, break down tags or VMs into batches of less than 2,000.
4. As an additional consideration, to avoid denial-of-service attacks to the tagging service, there is a limit of 360 tagging-related operations per second. To detect if you are hitting this limit, look for messages like "Request rejected due to high request rate. Try again later." in the vapi-endpoint log file (/var/log/vmware/vapi/endpoint/endpoint.log). If your creation code is exceeding this limit, we recommend inserting a pause of 2 ms or more between tag-creation calls.

4.3 Associate tag(s) with other objects

Associating tags with other objects (for example, hosts or datastores or content library items) has the same tradeoffs as associating tags with VMs, so the preceding discussion applies. As with the discussion on VMs, we have 4 cases to consider:

1. Associate 1 tag with 1 object (for example, host). We recommend using `attach(tagID, HostID)`
2. Associate 1 tag with multiple hosts. We recommend using `attachTagToMultipleObjects(tagID, HostID[])`
3. Associate multiple tags with 1 host. We recommend using `attachMultipleTagsToObject(tagID[], HostID)`
4. Associate multiple tags with multiple hosts. We recommend looping over hosts and calling `attachMultipleTagsToObject(tagID[], HostID)` for each host.

4.3.1 Query VMs associated with tags

To find the VMs associated with a tag, there are four APIs:

1. `List<TagAssociation.objectToTags> listAttachedObjectsOnTags(List<String> tagIDs)`: Given a list of tagIDs, this call returns the list of objects that are associated with these tags. The return value is a tuple that shows object id (as a DynamicID) and tag id. A DynamicID is an object that contains an identifier and a type. We give an example in the "Appendix."

2. `List<DynamicId> listAttachedObjects(tagID)`: Given a single tagID, this call returns a list of all objects (as Dynamic IDs) associated with this tag ID.
3. `List<TagAssocation.tagToObjects> listAttachedTagsOnObjects(List<DynamicID> objectIDs)`: Given a list of VM_IDs, this function returns all tags associated with those VMs. The return value is a tuple that shows the VM ID and its associated tag IDs.
4. `List<String> listAttachedTags(objectID)`: Given a VM, this call lists all tags associated with the VM.

Figure 8 below shows the latency to retrieve associated tags or associated objects for 1, 5, 10, or 15 ({1,5,10,15}) tags associated with 20,000 VMs for 3 APIs: `listAttachedObjects()`, `listAttachedObjectsOnTags()`, and `listAttachedTagsOnObjects()`. In figure 9, we do the same experiment using the `listAttachedTags()` API. We list it separately because its performance is much worse than the other approaches.

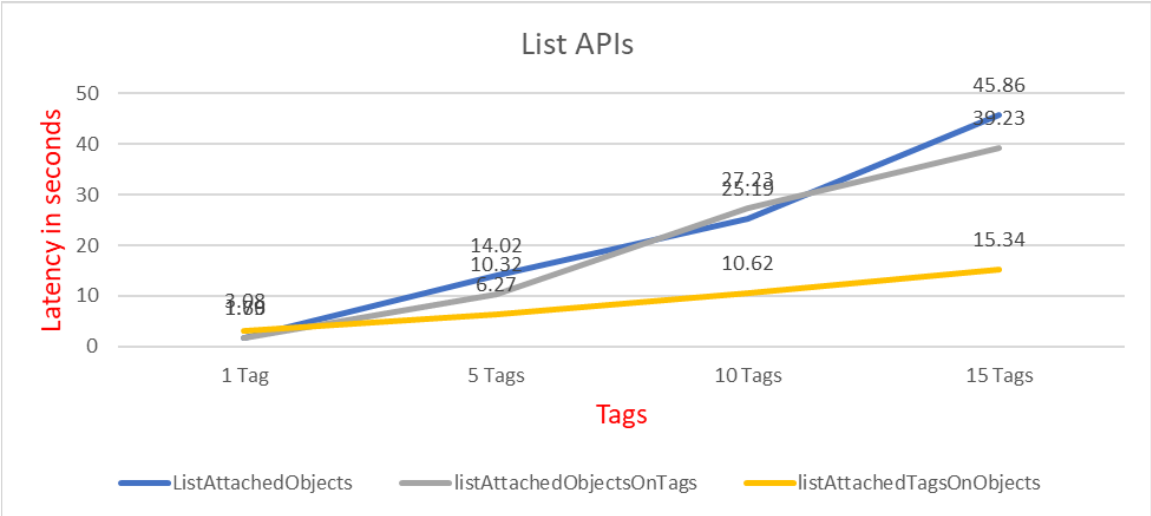

Figure 8: End-to-end time to retrieve associations for {1,5,10,15} tags attached to 20,000 VMs using various List APIs.

From figure 8 above, we see the following:

- The `listAttachedObjects()` call takes a single tag as an input. When we have 15 tags, we loop over the 15 tags. Each call returns 20,000 VMs. We have at most 15 iterations (one for each tag). The latency goes from 1.7 seconds to 45 seconds.
- In contrast, the `listAttachedObjectsOnTags()` call takes a list of tags and returns the objects attached to them. Each call again returns 20,000 VMs. In our experiments, we used 1 tag per call, rather than providing a list of tags, because when we provided 3 or more tags, the call failed due to internal buffer sizes. We have at most 15 iterations. Because we request tags using the same technique as `listAttachedObjects()`, the latencies are very similar.
- Finally, the `listAttachedTagsOnObjects()` call allows us to specify a list of objects. Because of this, we do not have to retrieve the tags for all 20,000 VMs at once. Instead, we break down the list of 20,000 VMs into batches of 2,000 VMs. For 1 tag, we provide 1 tag and iterate over the VMs in batches of 2,000, for a total of 10 iterations. For 15 tags, we provide an array of 15 tags, and again iterate over VMs in groups of 2,000, again for a total of 10 iterations to get associations for 20,000 VMs. This method scales the best, finishing more than 2x faster than the other approaches.

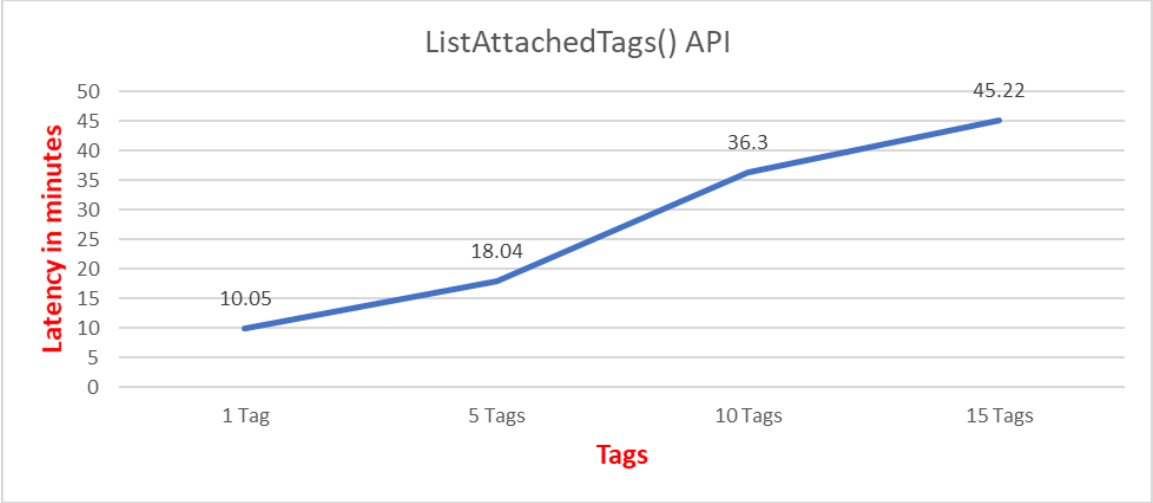

Figure 9: End-to-end time to retrieve associations for {1,5,10,15} tags attached to 20,000 VMs using `listAttachedTags()` API. `listAttachedTags()` is much slower than the other list calls, and we do not recommend using it in environments with more than 2,000 VMs.

From figure 9 above, we see that `listAttachedTags()` is much slower than the other approaches. This call only takes 1 object as an input and returns the appropriate number of attached tags. For 20,000 VMs, we need 20,000 iterations, which is substantially higher than for the other approaches.

We summarize the results in table 5 below.

Operation	1 tag associated with 20K VMs	5 tags associated with 20K VMs	10 tags associated with 20K VMs	15 tags associated with 20K VMs
`listAttachedObjects`	1.8 seconds	14 seconds	25 seconds	46 seconds
`listAttachedObjectsOnTags`	1.7 seconds	10 seconds	27 seconds	39 seconds
`listAttachedTagsOnObjects`	3.1 seconds	6.3 seconds	11 seconds	15 seconds
`listAttachedTags`	10 minutes	18 minutes	36 minutes	45 minutes

Table 5: Latency for various list APIs.

Because `listAttachedObjectsOnTags()` and `listAttachedObjects()` can return every object in the system, its performance is very sensitive to inventory size. Instead, if it is an option, we recommend using `listAttachedTagsOnObjects()`, since it allows you to break down the number of objects into smaller chunks and give more predictable performance per call.

It is important to point out that the list APIs above return IDs. Most client applications require names. To retrieve tag names from these IDs, you must use `get()` calls. For example, `Tag.get()` returns a `TagModel` object that has name, id, category_id, description, and used_by fields. For performance reasons, if it is practical, we recommend keeping a map of IDs to names on the client-side. Tag names are scoped to categories, so when storing a tag name, be sure to store the category name as well.

As a final example of batching, consider getting the mapping of all VMs associated with each tag. One way to do this is using `listAttachedObjects()`:

```
Get all categories
Get tags for each category
Get VMs associated with Tag using listAttachedObjects()
```

Because the number of objects associated with a tag may exceed 2,000, we recommend using `listAttachedTagsOnObjects()` instead. Because we do not know a priori how many tags will be associated with each object, to avoid overflowing internal vapi-endpoint buffers, one option is to loop over each object individually:

```
Get all objects using vSphere API for retrieving objects (List <DynamicId> objectList)
for (DynamicId object : objectList) {
   List<Tag.objectToTags> result = listAttachedTagsOnObjects(object)
}
```

We could have used either `listAttachedTagsOnObjects()` or `listAttachedTags()`, but we chose `listAttachedTagsOnObjects()` (iterating over each object individually) because the return value of `listAttachedTagsOnObjects()` is a mapping of tag to object. If we had used `listAttachedTags()`, we would have to manually create such a mapping.

The ***key takeaway*** from the discussion of the list APIs is that `listAttachedTagsOnObjects()` offers predictable performance and the most flexibility in retrieving associations. We recommend using this API and properly chunking the list of input objects for best performance when trying to retrieve tag associations.

5 Tagging APIs: PowerShell

The previous examples all used Java. In this section, we discuss using PowerShell. As described in Writing Performant Tagging Code: Tips and Tricks for PowerCLI [1], there are typically two ways to write PowerShell scripts for vSphere tags: either using cmdlets or using direct tagging APIs. For performance reasons, we recommend using direct tagging APIs instead of cmdlets, as indicated in the blog above.

The direct tagging service calls in PowerShell use the same vCenter server and tagging service resources as Java. Thus, the performance results are the same as the Java results above. In this section, we just focus on the difference in the API calls themselves. The API calls differ slightly from Java to PowerShell. Java uses camelCase, while PowerShell puts underscores in the method name. We give some examples below in table 6.

Description	Java	PowerShell (CIS service)
Attach one tag to one VM	`Attach()`	`Attach()`
Attach one tag to multiple objects	`attachTagToMultipleObjects()`	`Attach_tag_to_multiple_objects()`
Attach multiple tags to one object	`attachMultipleTagsToObject()`	`Attach_multiple_tags_to_object()`
List objects attached to a tag	`listAttachedObjectsOnTags()`	`List_attached_objects_on_tags()`

Table 6: Examples of differences between Java and PowerShell method names

There are also some minor differences in how arrays and objects are created between PowerShell and Java. Please see the examples in the "Appendix" for details.

PowerCLI version 11.5.0 is compatible with vCenter 6.0 through 6.7 U3. The documentation is here [6] and here [7].

6 Linked Mode

In Enhanced Linked Mode (ELM) [8], vCenter servers replicate tag and category information between peer nodes periodically. Each node replicates data with its peer every 30 seconds, so changes to categories and tag definitions on one node may take some time before they propagate to the other nodes in the ELM setup. The choice of topology can impact this replication time. For example, consider figure 10, which shows a simple ring topology [9] for vCenter servers.

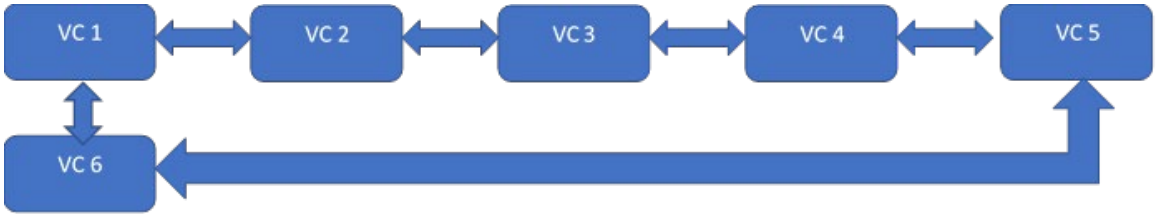

Figure 10: Ring topology for linked mode

In the figure above, the replication partners are as follows:

- VC 1: VC 2 and VC 6
- VC 2: VC 1 and VC 3
- VC 3: VC 2 and VC 4
- VC 4: VC 3 and VC 5
- VC 5: VC 4 and VC 6
- VC 6: VC 1 and VC 5

When a new tag is added on VC 1, here is a rough timeline of replication:

1. Time 0: tag added to VC 1
2. Up to 30 seconds later, VC 1 replicates to VC 2 and VC 6
3. Up to 30 seconds later, VC 2 replicates to VC 3; VC 6 replicates to VC 5
4. Up to 30 seconds later, VC 3 replicates to VC 4; VC 5 also replicates to VC 4, but ELM performs conflict resolution to ensure only one new tag definition is created on VC 4.

As the timeline above suggests, in this 6-node ring topology, a tag may take up to 90 seconds to replicate throughout the system. Because each node pushes changes at 30-second intervals, this is the worst-case latency. The nodes do not push changes at the same time, so in the best case, a tag change may occur right before that node has been scheduled to push changes. In this case, the changes would be reflected more quickly.

Different topologies will have different delays. For example, suppose VC 1 were NOT connected to VC 6, as shown below in figure 11.

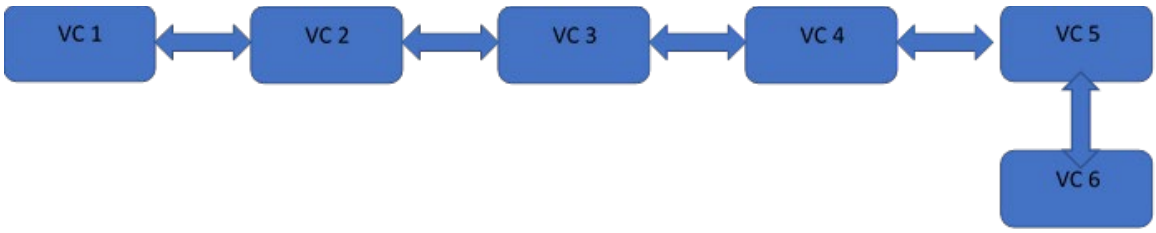

Figure 11: Linear topology for linked mode. We do not recommend this approach due to latency of tag/category replication.

1. Time 0: tag added to VC 1
2. Up to 30 seconds later: VC 1 replicates to VC 2
3. Up to 30 seconds later: VC 2 replicates to VC 3
4. Up to 30 seconds later: VC 3 replicates to VC 4
5. Up to 30 seconds later: VC 4 replicates to VC 5
6. Up to 30 seconds later: VC 5 replicates to VC 6.

As the timeline suggests, the worst case latency for the last node to receive a tag update is 30 seconds * 5 = 150 seconds after the tag has been added. The best case is nearly instantaneous, if the changes occur right before a node is scheduled to push tag updates.

7 Tags vs. Custom Attributes

There are some situations in which tags might not be the right fit. For example, suppose a user has 4,000 VMs in their vSphere environment, and they wish to assign an assetID to each VM. One option is to create a category called "assetID," and then have each tag within that category be a different assetID. However, this approach would generate numerous tags, which may cause performance issues. Another example might be assigning a "last power-off time" to each VM. One could have a category named "last power-off time" and have each tag be a different time, but this can again cause scalability issues. For situations like these two cases, it might be better to use vSphere custom fields and custom attributes. A discussion of custom fields and

custom attributes is beyond the scope of this paper: please refer to official VMware documentation [10] or William Lam's blog on this topic [11].

8 Scale Numbers for Good Performance

As mentioned in the Introduction, there are no hardcoded limits the number of categories, tags or tag associations that the tagging service can support. However, as we have shown in this paper, performance of the UI and APIs depends on the number of associations. In table 7, we list scale numbers that will work well *out-of-the-box* for vSphere 6.7. For these numbers of tags, categories, and associations, we expect acceptable out-of-the-box performance for the UI, second- and third-party applications, and custom scripts written by customers, as long as the scripts adhere to our best practices.

Target of Limit	Limit
Category definitions	6,000
Tag definitions	8,000
Tag Associations	25,000

Table 7: vSphere 6.7 tag limits for acceptable out-of-the-box performance

The limits above are based on response latency as well as hardcoded limits within vSphere processes to prevent denial-of-service (DoS) attacks and out-of-memory exceptions for large response bodies. We have shown in this paper how careful coding can allow a user to exceed these limits.

9 Concluding Remarks

In this paper, we have provided information about the proper use of tagging APIs for performance. We have discussed the following considerations:

Coding tips:

- For faster tag or category creation time, consider using multiple threads.
- It is faster to create tags spread across many categories, rather than putting all tags in a few categories.
- For faster assignment of tags to objects, consider using `attachMultipleTagsToObject()` or `attachTagToMultipleObjects()` instead of `attach()`.
- For faster listing of tag associations, consider `listAttachedObjectsOnTags()` or `listAttachedTagsOnObjects()`, rather than `listAttachedObjects()` or `listAttachedTags()`. The first two calls allow the user to specify a concrete list of tags or objects, and they allow chunking the requests to avoid overwhelming the tagging service.
- Linked mode topology can impact latency to replicate tags. We recommend a ring topology. For more details please see this blog [9].
- Keep a local map of IDs (tag IDs, category IDs, VM_IDs) to names so that you don't have to make a call to vCenter or the tagging service every time you need an ID from a name.

Important caveats:

- For attach calls, the maximum request and response size we recommend is 2,000 VMs or 2,000 tags. If possible, consider these limits when choosing which APIs calls to use.
- For listing associations, we recommend using `listAttachedTagsOnObjects()` and breaking down the list of objects into groups of 2,000.
- Consider using shorter tag names if you need to exceed 2,000 tags per attach call. The limit of 2,000 is based on the number of bytes in the response, not the sheer number of tags, so smaller tag names may allow you to retrieve more items in a single call.
- Creating and deleting tags creates tombstones in the tagging service database. These tombstones are needed for proper replication in linked mode, but excessive numbers of tombstones can impact performance. Please see KB 52387 [5] for more information.
- There are internal vCenter limits to avoid DoS attacks. The vCenter appliance allows no more than 360 requests per second to tagging and other RESTful vAPI-based services within the VCSA (for example, content library). If you are exceeding this limit in your environment, we recommend inserting a pause periodically to slow down the rate of requests. To know if you are hitting this limit, please check the vapi-endpoint log file (/var/log/vmware/vapi/endpoint/endpoint.log).
- In certain rare cases, you may need to increase the heap size of the vapi-endpoint service to accommodate high amounts of tagging traffic. When vapi-endpoint approaches its heap size, it triggers an alert within vCenter, so please be careful to check for this alert periodically.
- The latency to perform a tag association depends on the number of pre-existing associations. If many associations exist, it may be necessary to perform maintenance on the tag association table. The tag association table is a table within the main vCenter database. To perform such cleanup, please run the following vacuum job to clean up the database engine statistics:

  1. Connect to your VCSA VM via SSH as the root user.
  2. Log into postgres: `/opt/vmware/vpostgres/current/bin/psql -U postgres -d VCDB`
  3. Run vacuum analyze

```
1)  vacuum analyze cis_kv_keyvalue;
2)  vacuum analyze cis_kv_providers;
```

10 References

[1] Ravi Soundararajen and Joseph Zuk. (2019, June) Writing Performant Tagging Code: Tips and Tricks for PowerCLI. https://blogs.vmware.com/performance/2019/06/writing-performant-tagging-code-tips-and-tricks-for-powercli.html

[2] VMware. Java API for vSphere 6.5. https://vmware.github.io/vsphere-automation-sdk-java/vsphere/6.5.0/vsphereautomation-client-sdk/com/vmware/cis/tagging/TagAssociationStub.html

[3] VMware. vSphere Java APIs. https://github.com/vmware/vsphere-automation-sdk-java

[4] VMware. Package com.vmware.cis.tagging. https://vmware.github.io/vsphere-automation-sdk-java/vsphere/6.7.1/vsphereautomation-client-sdk/com/vmware/cis/tagging/package-summary.html

[5] VMware. (2018, November) Troubleshooting and addressing accumulation of tombstones in a Platform Services Controller (52387). https://kb.vmware.com/s/article/52387

[6] VMware. PowerCLI SDK. https://code.vmware.com/tool/vmware-powercli

[7] VMware. Tagging Service SDK (PowerCLI). https://code.vmware.com/docs/7335/powercli-11-0-0-user-s-guide/doc/GUID-F0088D49-CCFC-4F26-9C8C-845DE5AA951A.html?h=tags

[8] VMware. (2019, May) vCenter Enhanced Linked Mode for a vCenter Server Appliance with Embedded Platform Services Controller. https://docs.vmware.com/en/VMware-vSphere/6.7/com.vmware.vcenter.install.doc/GUID-7A90361C-E95B-4C53-A328-3DB9EE20536D.html

[9] Emad Younis. (2018, May) vCenter Server Architecture Part 1 – The Basics. https://emadyounis.com/vcenter-server-architecture-part-1-the-basics/

[10] VMware. (2019, May) Custom Attributes in the vSphere Web Client. https://docs.vmware.com/en/VMware-vSphere/6.7/com.vmware.vsphere.vcenterhost.doc/GUID-73606C4C-763C-4E27-A1DA-032E4C46219D.html

[11] William Lam. (2015, January) Custom Attributes != vSphere Tags. https://www.virtuallyghetto.com/2015/01/custom-attributes-vsphere-tags.html

[12] VMware. VMware vSphere Automation SDK for Java. https://github.com/vmware/vsphere-automation-sdk-java

About the authors

Raju Angani is a staff performance engineer in the Performance team at VMware. His main areas of focus are vSphere tagging performance and vCenter database performance.

Ravi Soundararajan is a principal engineer in the Performance team at VMware, specializing in vCenter performance and scalability. He has presented numerous times about vCenter at VMworld. His twitter handle is @vCenterPerfGuy. Ravi received his SB from MIT and his MS and PhD degrees from Stanford, all in Electrical Engineering.

Maarten Wiggers is a staff II engineer and works on vCenter with a focus on tagging, policies, and resource management. He has been with VMware for 6 years and has a PhD from the University of Twente, The Netherlands.

Acknowledgements

We gratefully acknowledge Joseph Zuk, Oleg Zaydman, Sangeeta Jogeklar, Abhijith Marulaiah Prabhudev, Kyle Ruddy, Jake Robinson, Chungyen Chang, Yong Li, Udaya Ganiga, and Vishwa Srikaanth for their assistance with the content of this paper.

11 Appendix

11.1 Experimental setup and software versions

We used VMware PowerCLI 10.1.0 build 8346946. Note that newer versions may demonstrate better performance.

We used Java 1.8 ("1.8.0_231").

Our vCenter was vCenter 6.7 U3 running on ESXi 6.5 U2. Our inventory was composed of 2,000 hosts and 25,000 VMs, simulated using a proprietary VMware-internal tool. Because of the large number of VMs, we used the "Extra Large" size for the VCSA.

11.2 Java code examples

Note: some of these examples come from the following Github repository: https://github.com/vmware/vsphere-automation-sdk-java [12].

For each of the Java examples below, we will need a set of imports. Here are the imports we used in our code snippets:

```
import vim25
import com.vmware.cis.tagging.Category;
import com.vmware.cis.tagging.CategoryTypes;
import com.vmware.cis.tagging.Tag;
import com.vmware.cis.tagging.TagAssociation;
import com.vmware.cis.tagging.TagModel;
import com.vmware.cis.tagging.TagTypes;
import com.vmware.cis.tagging.CategoryModel.Cardinality;
```

11.2.1 Example J0: VM DynamicID and Tag TagID

Many code samples involve the use of VM DynamicID objects. A DynamicID object is similar to a Managed Object Reference (MoRef, described in [10]), but the identifier and type are stored in separate fields. Here is a simple code snippet for constructing a DynamicID from a VM MoRef. We first retrieve the VM by name, and then we get its MoRef. Finally, we construct a DynamicID from this MoRef.

```
// Retrieve the VM MoRef from its name
this.VMMoRef = VimUtil.getVM(this.vimAuthHelper.getVimPort(),
                             this.vimAuthHelper.getServiceContent(),
                             "VM_NAME");
assert this.VMMoRef != null;
// convert the MoRef to vAPI DynamicID
this.vmDynamicId = new DynamicID(this.VMMoRef.getType(),this.VMMoRef.getValue());
```

11.2.2 Example J1: Retrieving category/tag IDs from category/tag names

Most tagging APIs require tag IDs. A tag ID is globally unique, while a tag name is unique within a category. To find a tag ID given a tag name, you must also specify the category ID.

To following code finds a category ID given a category name:

```
Category _categoryService = vapiAuthHelper.getStubFactory().createStub(Category.class,
                                                              sessionStubConfig);

// Get CategoryId
public String getCategoryId(String catName) {
  List<String> catIds = _categoryService.list();
  String retCatId = NULL;
  if (catIds.size() > 0) {
    for (String cat : catIds){
       if (_categoryService.get(cat).getName().equals(catName)) {
          retCatId = _categoryService.get(cat).getId();
       }
    }
  }
  return retCatId;
}
```

The following code gets a tag ID given a tag name and a category name, using the routine above to map the category name to an id.

```
Tag _taggingClient = vapiAuthHelper.getStubFactory().createStub(Tag.class,
                                                              sessionStubConfig);

// Get TagId for a given tagName and CategoryName
public String getTagId(String tagName, String catName) {
   List<String> tagIds = _tagProvider.list();
   String catId = getCategoryId(catName);
   String retTagId = "";
   if (tagIds.size() > 0) {
      for (String tag : tagIds) {
         if (_tagProvider.get(tag).getName().equals(tagName)
&&_tagProvider.get(tag).getCategoryId().equals(catId)) {
            retTagId = _tagProvider.get(tag).getId();
         }
      }
   }
   return retTagId;
}
```

11.2.3 Example J2: Create Category

```
/**
 * Creates a tag category
 *
 */
private String createTagCategory(String name, String description,
                                 Cardinality cardinality, Set<String> AssociableTypes) {
    CategoryTypes.CreateSpec createSpec = new CategoryTypes.CreateSpec();
    createSpec.setName(name);
    createSpec.setDescription(description);
    createSpec.setCardinality(cardinality);
    createSpec.setAssociableTypes(associableTypes);
    return this.categoryService.create(createSpec);
}
// Sample invocation: create category named "GuestOS"
Set<String> associableTypes = new HashSet<String>(); // empty hash set
associableTypes.add("VirtualMachine");
String categoryId = createTagCategory("GuestOS",
                                      "Guest OS",
                                      CategoryModel.Cardinality.SINGLE,
                                      associableTypes);
```

11.2.4 Example J3: Create Tag

In this example, we use the categoryId from "Example J2: Create Category" example above.

```
/**
 * Creates a tag
 *
 */
private String createTag(String name, String description,
                         String categoryId) {
    TagTypes.CreateSpec spec = new TagTypes.CreateSpec();
    spec.setName(name);
    spec.setDescription(description);
    spec.setCategoryId(categoryId);

    return this.taggingService.create(spec);
}

// Sample invocation: create tag named "Windows 10" in categoryId from above
String tagID = createTag("Windows 10","DB Operating System",categoryId);
```

11.2.5 Example J4: Associating a tag with a VM (attach())

In this example, we use the VM DynamicID (vmDynamicId) from example J0 above and the tagID from example J3 above.

```
/**
 * Associates tagId from Example J3 to VM whose DynamicID is vmDynamicId (Example J0)
 * We assume vmDynamicId is retrieved as shown in Example J0 above.
 *
 */

// Get a handle to the tagAssociation Methods
tagAssociation = vapiAuthHelper.getStubFactory().createStub(TagAssociation.class,
                                                            sessionStubConfig);

tagAssociation.attach(tagId,vmDynamicId);
```

11.2.6 Example J5: Associating a tag with multiple VMs (attachTagToMultipleObjects())

Attach 1 tag to 1,000 objects

```
// Attach 1 tag to first 1,000 VMs
// Assume we have a list of VM DynamicIDs ➔ List<DynamicID> vmDynamicIds

int totalVMs = 1000;
List<DynamicID> vmDynamicIdsNeeded = new LinkedList<>();
for (int k = 0; k < totalVMs; k++) {
   vmDynamicIdsNeeded.add(vmDynamicIdsNeeded.add(vmDynamicIds.get(k));
}

// Get a handle to the tagAssociation Methods
tagAssociation = vapiAuthHelper.getStubFactory().createStub(TagAssociation.class,
                                                            sessionStubConfig);
tagAssociation.attachTagToMultipleObjects(tagId,vmDynamicIdsNeeded);
```

11.2.7 Example J6: Associating multiple tags with multiple VMs (attachMultipleTagsToObject())

Attach 10 tags to 1,000 objects

```
// Attach 10 tags to 1,000 VMs
// Assumes we have a list of all tagIDs ➔ List<String> tagIds
// Assumes we have a list of all VM_IDs ➔ List<DynamicId> vmDynamicIds
// We will assign the first 10 of these tags to 1,000 VMs

int totalVMs = 1000;
int totalTags = 10;

// Get first 10 tags from the total tagIds
List<String> associableTags = new LinkedList<>();
for (int k = 0; k < totalTags; k++) {
   associableTags.add(tagIds.get(k));
}

// Get first 1,000 VMs from the total VM_IDs
List<DynamicID> vmDynamicIdsNeeded = new LinkedList<>();
for (int k = 0; k < totalVMs; k++) {
   vmDynamicIdsNeeded.add(vmDynamicIds.get(k));
}

tagAssociation = vapiAuthHelper.getStubFactory().createStub(TagAssociation.class,
                                                            sessionStubConfig);
for (int i = 0; i < totalVMs; i++) {
  tagAssociation.attachMultipleTagsToObject(vmDynamicIdsNeeded.get(i), TagIds);
}
```

11.2.8 Example J7: List tags associated with a VM

```
// List tags associated with vm vmDynamicId from above.
// Assumes tagAssociation object from above
List<String> retTagIds = tagAssociation.listAttachedTags(vmDynamicId);
```

11.2.9 Example J8: List VMs associated with a tag

```
// List tags associated with tagId from above
// Assumes tagAssociation object from above
List<DynamicID> retDynamicIds = tagAssociation.listAttachedObjects(tagId);
```

11.2.10 Example J9: List tags associated with a group of VMs

```
// Helper object for storing results
ObjectToTags {
    DynamicID objectId;
    List<String> tagId;
}

// List tags associated with 10 VMs from vmDynamicIdsUsed list above
// Assumes tagAssociation object from above
List<TagAssociationTypes.ObjectToTags> retObjectToTags =
    tagAssociation.listAttachedTagsOnObjects(vmDynamicIdsUsed.subList(0,10));
```

11.2.11 Example J10: List VMs associated with a group of tags

```
// Helper object for storing results
TagToObjects {
    String tagId;
    List<DynamicID> objectIds;
}
// Assumes we have a list of all tagIDs ➔ List<String> tagIds
List<TagAssociationTypes.TagToObjects> retTagToObjects =
    tagAssociation.listAttachedObjectsOnTags(tagIds);
```

11.2.12 Example J11: Retrieving tag information (e.g., name, description) from the tag ID

```
// TagModel contains name, description, tag ID, category ID, and usedBy fields
TagModel retTagModel = _tagProvider.get(tagID);
```

11.3 C. PowerShell code examples

In this section, we give select examples of PowerShell. Examples of creating a category, creating a tag, attach(), attach_ tag_ to_ multiple_ objects(), and attach_ multiple_ tags_ to_ object() are given in the Performance Team blog on tagging [1] and are therefore not repeated here.

11.3.1 Example P0: Creating VMID Objects and retrieving Tag IDs

```
# Retrieving a tag ID from its name
function Get-TagIdFromName {
    Param ($a)
    $allTagMethodSvc = Get-CisService com.vmware.cis.tagging.tag
    $allTags = @()
    $allTags = $allTagMethodSvc.list()
    foreach ($tag in $allTags) {
        if (($allTagMethodSvc.get($tag.value)).name -eq $a) {
            return $tag.value
        }
    }
    return $null
}

# Creating a single VM ID Object
$testVM = Get-VM -Name "myVM"

$VMInfo = $testVM.ExtensionData.MoRef
$vmid = New-Object PSObject -Property @{
    id = $VMInfo.value
    type = $VMInfo.Type
}

# Create an array of VM IDs
$allVMs = Get-VM
$useTheseVMIDs = @()

# Create VM objects for all VMs.
# This should be done once for all VMs, not every time
# you want to do an association.
for ($i = 0; $i -lt 1000; $i++) {
    $VMInfo = $allVMs[$i].extensiondata.moref
    $useTheseVMIDs += New-Object PSObject -Property @{
        id = $VMInfo.value
        type = $VMInfo.type
    }
}
```

11.3.2 Example P1: List tags associated with a VM

```
$allCategoryMethodSVC = Get-CisService com.vmware.cis.tagging.category
$alltagMethodSVC = Get-CisService com.vmware.cis.tagging.tag
$allTagAssociationMethodSVC = Get-CisService com.vmware.cis.tagging.tag_association

# Use first VM from above list
$useThisVMID = $useTheseVMIDs[0]
$tagList = $allTagAssociationMethodSVC.list_attached_tags($useThisVMID)
```

11.3.3 Example P2: List VMs associated with a tag

```
$allTagMethodSvc = Get-CisService com.vmware.cis.tagging.tag

# Use method above to get tag ID from tag name
$tagId = Get-TagIdFromName "myTag"
$vmList = $allTagAssociationMethodSVC.list_attached_objects($tagId)
```

11.3.4 Example P3: List tags associated with a group of VMs

```
$assocSVC = Get-CisService com.vmware.cis.tagging.tag_association

# Pass in VMs in groups of 2,000
$sampleSize = 2000

# Assume a list of VMs: $useTheseVMIDs
$index = 0
$tagList = @()
for ($i = 0; $i -le ($useTheseVMIDs.Count/$sampleSize); $i++) {
    $tagList += $assocSVC.list_attached_tags_on_objects($useTheseVMIDS[$index..($index +
$sampleSize)])
}
```

11.3.5 Example P4: List VMs associated with a group of tags

```
$assocSVC = Get-CisService com.vmware.cis.tagging.tag_association

# Loop over 15 tags and get the result.
# If there are too many VMs and you pass in all 15 tags in one go, you may get a RESPONSE error
$tagIdList = $tagList[0 .. 15]
$VMAssocList = @()
foreach ($tag in $tagIdList){
    # Must use a list because list_attached_objects_on_tags assumes a list of tags as an input
    # We are doing this one at a time, but you can use multiple at a time as long
    # as you do not get a RESPONSE error
    $useTheseTags = @()
    $useTheseTags += $tag
    $VMAssocList += $assocSVC.list_attached_objects_on_tags($useTheseTags)
}
```